P.S.I.

LANGAGES

LE LIVRE DE TURBO-C SUR PC ET COMPATIBLES

PIERRE BRANDEIS
FRANCIS PIEROT

PROGRAMMATION

- Installation
- Pointeurs
- Fichiers

Connaissez-vous les autres ouvrages P.S.I. sur les langages ?

Langage C
Entrées-sorties en C - Jean de Brabandt
Programmer en C - Claude Nowakowski
C sur Atari ST - Claude Nowakowski
C et ses fichiers - Jacques Boisgontier et Jean-Pierre Lagrange
Clefs pour C - François Piette
Bibliothèque mathématique en C - Claude Nowakowski

BASIC
Le Basic et ses fichiers - Jacques Boisgontier
Le livre du Basic sur PC et compatibles - Jacques Boisgontier

Turbo-Pascal
Turbo-Pascal et ses fichiers - Jacques Boisgontier et Christophe Donay
Turbo-Pascal sur PC et compatibles - Frédéric Blanc et Pierre Brandeis
Turbo-Pascal pas à pas - Jean-Michel Gaudin
Clefs pour Turbo-Pascal sur PC et compatibles - Frédéric Blanc et Pierre Brandeis
Programmer en Pascal - Daniel-Jean David et Jean-Luc Deschamps

Cobol
Cobol sur PC et compatibles - Daniel-Jean David et Daniel Trécourt

Prolog
Le livre de Turbo Prolog - Michel Treillet
Turbo Prolog pas à pas - Michel Treillet
Le langage D-Prolog - Philippe Donz et Rosalie Hurtado
Le sytème Prolog - version 2.2. - Pierre Giraud

Pour connaître les dernières nouveautés P.S.I.,
ou nous soumettre un problème technique,
nous mettons à votre disposition un service Minitel

Service Minitel

Sur le 3615 tapez OI puis PSI

© Éditions P.S.I., une société du groupe CEP Communication
5 place du Colonel Fabien 75491 Paris Cédex 10
1987
ISBN: 2-86595-467-6

LANGAGES

LE LIVRE DE TURBO-C SUR PC ET COMPATIBLES

PIERRE BRANDEIS
FRANCIS PIEROT

ÉDITIONS P.S.I.
1987

SOMMAIRE

INTRODUCTION

ET AVERTISSEMENT

ENFIN UN LIVRE COMPLET SUR TURBO-C ?

Eh bien non !

Lorsque l'on connaît la richesse extraordinaire du langage C, il paraît évident qu'elle ne permet pas, matériellement, d'en atteindre la quintessence en un seul ouvrage, aussi énorme soit-il. Il y a beaucoup trop de choses à voir, à apprendre, à maîtriser. La quantité d'informations dispensées sur le langage C dans d'autres ouvrages est déjà assez extraordinaire en soi, mais il suffit de jeter un œil sur l'épaisseur des manuels originaux de Turbo C pour constater l'ampleur de la tâche : et encore, ne s'agit-il guère, dans les manuels Borland, que de guider le lecteur débutant.

Nous avons donc choisi, dans cet ouvrage, d'aborder le Turbo C sous un aspect initiateur, en effectuant notamment quelques parallèles avec Turbo Pascal lorsque cela paraissait approprié, non pas pour plagier la documentation de Borland, mais parce que ces deux langages ont de nombreux points communs. Cela devrait permettre un premier contact favorable avec C, ou une transition sans douleur pour les adeptes -nombreux- de Turbo Pascal.

Ce livre ne vous emmènera donc pas au sommets du langage.

Nous avons délibérément choisi une approche très "synthétique", visant plus à vous faire découvrir ce qui est vraiment utile au début, ce qui vous sera indispensable et ce que vous devrez savoir pour aller plus loin. Mais vous n'apprendrez pas dans ces pages comment interfacer Turbo C avec Prolog ou de l'assembleur, contrôler l'environnement MS-DOS, changer de modèle de mémoire, ou autres manipulations critiques. Cela, ce sera pour plus tard.

Nous avons donc volontairement limité le niveau des connaissances requises pour attaquer la lecture de cet ouvrage. Le nombre de pages étant limité, il est évident que le niveau atteint le sera aussi. Néanmoins, lorsque vous aurez achevé la lecture de cet ouvrage vous en saurez bien suffisamment pour écrire des applications assez complexes, ou vous plonger dans la lecture d'ouvrages plus ésotériques et mystérieux.

UNE MURAILLE HISTORIQUE S'ECROULE!

Le langage C a longtemps été l'apanage exclusif des minis et gros ordinateurs. Le TRS-80 Modèle 1 eut l'honneur (il y a déjà 8 ans), de disposer d'un compilateur Lattice, mais programmer en langage C avec 32 Ko de mémoire était en fait un véritable suicide mental !

En raison des limitations des micro-ordinateurs et du manque de ressemblance entre leurs systèmes d'exploitations et Unix, le mur a donc toujours semblé indestructible. De plus, le C est depuis sa naissance l'exemple même du langage réservé à "ceux qui savent"; l'inaccessible, l'intouchable, l'obscur et principal bastion du prestige de la mini.

De nos jours, l' affrontement psychologique entre partisans de la mini et de la micro n'est plus de mise face au progrès technologique. Un compatible AT équipé de 640 Ko de mémoire et d'un disque dur 20 Mo coûte le prix d'un Apple II (64 Ko) avec un lecteur de disquettes (143 Ko) il y a cinq ans. Face à cette évolution, il était inévitable qu'un jour la barrière C s'effondre enfin.

Mais implanter C sur un micro-ordinateur n'était pas suffisant ! Chacun sait que les micros sont bien faibles face à leurs confrères minis et gros calculateurs. Mais ils ont un avantage incontestable et incontesté : un seul utilisateur y travaille à la fois (on évite généralement de travailler en multi-utilisateurs sur un micro de développement). Le système peut donc être totalement dédié à sa tâche de compilation ou d'édition, d'où une rapidité de fonctionnement de plus en plus spectaculaire face à celle des minis.

Or, si l'on conçoit qu'il faille une minute pour compiler un programme C de cent lignes sur une machine qui gère 10 ou 20 utilisateurs simultanés, cela paraît difficilement acceptable sur un micro, où le développeur est tout seul! Et jusqu'à fin 1986, il fallait pourtant se résoudre à l'évidence : aucun compilateur C du commerce, sur aucune machine, n'était capable d'avoir un comportement acceptable. Entre les compilateurs qui saturaient la mémoire au delà de 1000 lignes, ceux qui compilaient environ dix lignes à la minute, et ceux qui généraient systématiquement des bugs, le choix était simple. Seul le compilateur Microsoft version 3 affichait une bonne fiabilité, au prix hélas d'une lenteur exaspérante.

Et soudain, aux environs de septembre 1986, la révolution est arrivée. Comme Turbo Pascal 3.0, Turbo C a pulvérisé d'un coup les performances de tous les compilateurs concurrents qui existaient. On peut d'ailleurs considérer que, tant que Microsoft n'avait pas achevé la version 4 de son compilateur, Turbo C n'avait absolument aucun concurrent!

La première caractéristique de ce compilateur, c'est sa rapidité exceptionnelle. La plupart des compilateurs C classiques prennent tranquillement leur temps. Turbo C compile environ 10000 lignes à la minute sur un compatible AT.

La seconde particularité de Turbo C est bien entendu le légendaire "effet Turbo", avec environnement intégré. Ce dernier simplifie considérablement des manipulations qui, d'ordinaire, se font sous DOS à l'aide de longues lignes de commandes. Avec Turbo C, deux ou trois appuis de touches suffisent pour effectuer la plupart des opérations!

L'arrivée sur le marché d'un tel compilateur remet bien entendu beaucoup de vérités en question. L'opinion la plus répandue est que "le C n'est pas fait pour les débutants". Mais est-ce vraiment une raison pour en limiter l'accès aux

"pros" de l'informatique ? De toutes façons, l'utilisation de Turbo C est tellement simple, que cette assertion n'a plus aucune valeur. Turbo C donne justement l'opportunité à de nombreux adeptes du Basic ou de Turbo Pascal de passer à un langage qui, malgré sa réputation, n'est pas si difficile d'accès qu'on le laisse entendre. Turbo C gomme littéralement l'aspect compliqué de la mise en œuvre du C. Il ne reste à maîtriser que l'essentiel, le langage lui-même (qui est assez proche du Pascal au niveau organisation, mais qui s'en éloigne par de nombreux points de détails).

Bien entendu, Turbo C reste dans la lignée des produits Borland les plus récents. Il rend la programmation aisée, mais ne réduit pas les possibilités du langage (certaines limitations de Turbo Pascal 3.0 durement critiquées feront parfois préférer Turbo C). Contrairement à Turbo Pascal 3.0, il peut être utilisé comme un compilateur C classique (pour réaliser des applications professionnelles) sans passer par l'environnement intégré. Cela permet de travailler d'une façon plus habituelle, avec Link séparé. Cela permettra aussi (et surtout, serions nous tentés de dire!) de mixer des modules réalisés dans différents langages : Turbo Prolog, Turbo C, Turbo Pascal 4.0, assembleur ... Livré avec d'abondantes librairies, il propose six modèles de gestion de la mémoire, un choix de l'optimisation (que l'on peut régler en fonction de divers paramètres), et de nombreuses options pour les programmeurs avertis, permettant de contrôler la compilation et la génération de code d'une façon assez complète. Les fonctions des librairies sont innombrables, ce qui fait de Turbo C un langage riche, évolutif, et particulièrement performant car très fourni. Son prix particulièrement attractif ne nuit bien entendu pas à la qualité de l'ensemble. En bref, c'est du Borland!

IMPORTANT :

CET OUVRAGE N'EST PAS DESTINE AUX POSSESSEURS DE TURBO C QUI AURAIENT "EGARE" LEUR DOCUMENTATION.

Et d'ailleurs il ne la remplace pas.

Si vous possédez une copie non légale (c'est à dire que vous n'avez pas acheté l'original), veuillez s'il vous plaît lire ce qui suit.

Nous voudrions mettre en garde les lecteurs utilisant une version piratée ou copiée du compilateur, car il y en a sans doute. Les autres peuvent lire ce qui suit mais ils ne sont bien entendu pas concernéscest remarques !

POSSESSEURS DE COPIES , ATTENTION :

Nous considérons que pirater un logiciel de Borland (ou d'un autre éditeur, d'ailleurs) est réellement de la mesquinerie. Etant donnés les efforts de Borland pour justement mettre Turbo C à la portée financière de tout le monde, nous ne voyons vraiment pas ce qui pourrait excuser son piratage ! De plus, pour avoir

été nous même auteurs de logiciels, nous connaissons sans doute trop bien le problème du piratage pour l'accepter.

L'informatique, est une science, c'est certain, mais pas tout à fait une science exacte (sinon les bugs n'existeraient pas, et il n'y aurait qu'un seul langage). Le coté "artistique" de la conception d'un logiciel apparaît spontanément à tous ceux qui s'adonnent à cette activité. Aussi, considérons-nous que la mise au point d'un logiciel comme Turbo-C tient autant de l'exploit purement technique que d'une extraordinaire performance artistique et humaine. Ce style de travail doit se respecter, à notre avis, et mérite la plus haute estime.Copier et diffuser une telle œuvre revient à mépriser le travail des concepteurs et des programmeurs .

Si vous êtes de ceux qui utilisent des logiciels (en particulier Turbo C) piratés ou copiés que vous auriez sans doute acheté, reconsidérez votre point de vue : certes, votre copie ne vous a pas coûté cher, mais le "plaisir" de posséder un emballage, une documentation et souvent le suivi de l'éditeur (mises à jour gratuites, support technique, etc...) ne se monnaye pas.

D'autant que la documentation de Borland est pratiquement indispensable comme ouvrage de référence ultime. Le guide utilisateur et le manuel de référence de Turbo C sont indispensables pour qui veut réellement utiliser ce compilateur. Et le support téléphonique Borland n'est pas non plus un service négligeable (accessible uniquement à ceux qui possèdent leur numéro de série).

Mais après tout, le thème de l'achat ou non des logiciels pose essentiellement des problèmes de conscience. Certains n'éprouvent pas le moindre remord en utilisant quotidiennement le travail d'autres personnes sans payer leur dû, d'autres estiment au contraire que tout travail mérite rétribution et achètent les outils ou programmes dont ils ont besoin.

PIRATES, ATTENTION :

En dehors de cet aspect moral des choses, vous devez savoir que le seul fait de posséder une copie non autorisée d'un logiciel (c'est à dire si vous ne pouvez pas produire son original) vous fait tomber sous le coup de la loi du 3 Juillet 1985 . Et la loi en question vous assimile à un FAUSSAIRE, ou CONTREFACTEUR. A savoir, vous risquez de trois mois à deux ans de prison, et de 6000 à 120000 francs d'amende, plus certainement confiscation de tout matériel informatique.

A bon entendeur...

CHAPITRE1

QU'EST-CE QUE TURBO C ?

Avant de nous pencher sur Turbo C proprement dit, il est évident qu'il nous faut déjà découvrir le C lui-même. Car C possède une "longue" histoire. Comme nous l'avons expliqué dans l'introduction, il existait sur mini-ordinateurs bien avant d'accéder à la micro-informatique.

UNE HISTOIRE D'AMOUR LOGICIELLE !

Le C existe dans sa version quasi-actuelle depuis 1971/72 sur les minis. Il a été conçu par Dennis Ritchie, des Laboratoires Bell, afin de doter son système d'exploitation UNIX d'un langage de développement. Il faut savoir qu'à l'époque, les langages évolués étaient rarement capables de traiter des applications "système". La programmation des systèmes d'exploitation passait généralement par l'assembleur, faute de langage approprié. Ritchie conçut donc le premier C en se basant sur un autre langage BCPL, qui existait déjà au sein de Bell. Par la suite, Dennis Ritchie et Brian Kernighan écrivirent le compilateur C et l'ensemble des utilitaires existant sous UNIX en C (y compris Unix lui-même plus tard), ce qui les rendit portables. Depuis ce jour, C et Unix marchent main dans la main. Il a fallu une dizaine d'années avant que l'on pense à séparer C d'Unix, tant ils semblaient dépendants l'un de l'autre.

En réalité, si l'on excepte le fait qu'ils sont nés d'un même auteur (Ritchie) et qu'ils ont évolués de concert, il n'y a pas de liaison particulière entre le langage C et le système d'exploitation Unix. La légende associe inévitablement les deux essentiellement parce que le C, tel que nous le connaissons, à été mis au point pour Unix, puis développé, étendu et enrichi de nouvelles possibilités sous ce même système, au fur et à mesure que Unix progressait lui-même. A part cela, C est un langage évolué et, par sa définition même, les programmes sont théoriquement indépendants du système d'exploitation et de la machine qui les accueille.

L'arrivée sur le PC d'un compilateur comme Turbo C est bien l'occasion de constater que le couple "UNIX-C" n'est plus figé pour l'éternité dans sa dualité. Si Unix restera probablement toujours le système "avec le langage C", il est particulièrement évident que C va progressivement se répandre à tous les étages de l'Informatique, grâce à la richesse de son environnement.

Le langage C bénéficie depuis peu d'une norme ANSI. Cela signifie qu'une bonne partie de ses caractéristiques est "normalisée" et doit être respectée par les prochains compilateurs (Turbo C fait partie de cette nouvelle génération de compilateurs et respecte donc la norme ANSI). L'ANSI n'a normalisé le C que très récemment, presque quinze ans après sa création (Pascal en avait

bénéficié très rapidement). Conséquence de cette adoption tardive : la norme a profité de toute l'expérience accumulée sur C, et remédie dans ses spécifications à la plupart des défauts majeurs du langage. A priori, les compilateurs comme Turbo C possèdent donc un avantage considérable puisqu'ils arrivent sur l'IBM PC en bénéficiant de quinze ans d'expérience des utilisateurs du langage C.

QU'EST-CE QUE LE C?

Le C a une caractéristique principale : la compacité. Il permet de réaliser des programmes extrêmement courts pour une tâche parfois complexe. On serait même tenté de dire que, dans l'absolu, le C est difficilement lisible tant il est compact. Ça, c'est pour son aspect visuel.

Le C est un langage "compilé" et "linké". Cela signifie que le programme en C proprement dit, avant d'être utilisable par l'ordinateur, doit, dans une première étape, être traduit en un codage plus accessible au processeur (c'est le "code objet"); les ressources propres à l'ordinateur utilisé (écran, lecteur de disquettes, clavier ...) doivent ensuite être reliées à ce code intermédiaire (c'est le "LINK"), ce qui permet de générer un code dit "exécutable", que l'ordinateur est alors capable effectivement d'exécuter.

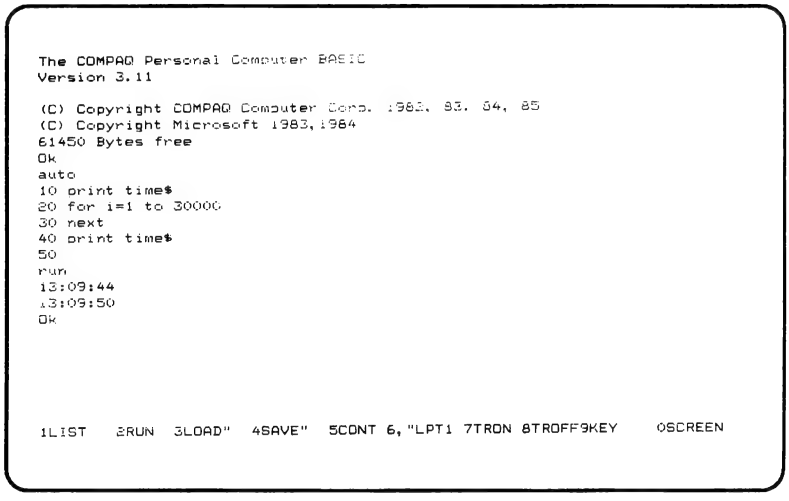
Bien entendu, sous Turbo C, toute cette manipulation (compilation, link) est transparente. En cela, le C se rapproche des autres langages compilés- linkés (Turbo Pascal 4.0, par exemple, qui doit sans doute être disponible à l'heure où vous lisez ces lignes), et s'éloigne, en apparence seulement, des compilateurs classiques pour lesquels toute une série de manipulations et de transferts est nécessaire.

Un programme en langage C est appelé "source". Il s'agit d'un fichier texte tout à fait anodin, tel que vous pourriez en créer avec l'utilitaire EDLIN de MS-DOS, par exemple. Ce n'est qu'une suite de mots, de signes, que l'ordinateur est bien incapable de comprendre, tant qu'on ne l'a pas compilé et linké. Mais le C étant un langage dit "évolué", comme le Pascal, son charabia est par contre assez compréhensible par le programmeur qui, ne l'oublions pas, est humain. C propose donc tout un ensemble d'entités à la disposition du programmeur pour que celui-ci n'ait pas à se préoccuper du processeur, de la mémoire, du disque dur ou pire encore.

INTERPRETEURS ET COMPILATEURS

Pour bien saisir la nuance entre un langage interprété (comme le GWBASIC ou BASICA livré avec le PC-DOS) et un langage compilé (comme Turbo C ou Turbo Basic), voici un exemple très simple.

Programmons en Basic les lignes suivantes, qui réalisent une simple boucle exécutée 30 000 fois :



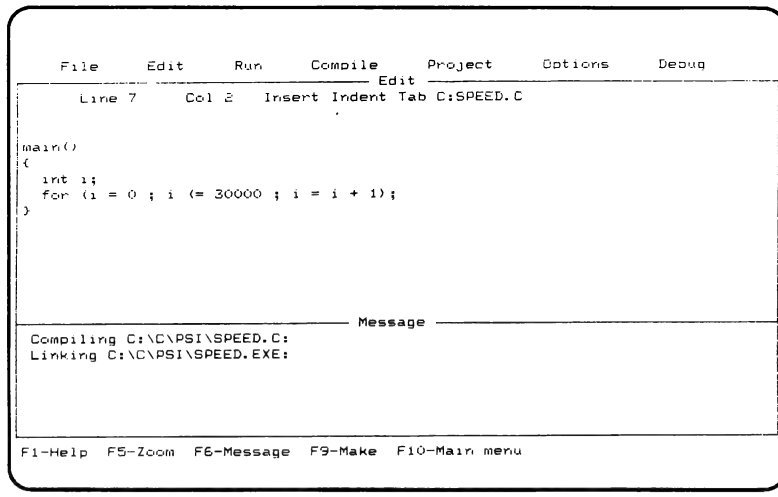
Un programme Basic tout simple

Lorsque nous demandons RUN pour exécuter ce programme, nous passons en fait la main à un programme spécialisé, "l'interpréteur". Celui-ci attaque la ligne 20 et décode l'instruction qui s'y trouve. Il crée la variable I, y range la valeur de départ (soit 1) et vérifie que la valeur finale est du bon type (si c'est une chaîne, il y a erreur).

Puis, la ligne20 étant terminée, il passe à la ligne 30. Là, il tombe sur "NEXT I". Il recherche donc la variable I, et y récupère les informations sur la boucle. Cela lui permet de retourner à la ligne 20, de faire un test pour savoir si I est inférieur à 30 000, et de passer selon le cas soit en ligne 30, soit à la suite de l'instruction NEXT.

Le problème, c'est que ces opérations sont bien entendu effectuées en langage machine. L'interpréteur ne se souvient pas, en arrivant à la ligne 30, qu'il l'a peut-être déjà effectuée. Il retraduit donc de nouveau l'instruction NEXT I, recherche ensuite une nouvelle fois la variable I, et recherche de nouveau la ligne 0, etc...

En Turbo C, le programme devient le suivant :



Le même programme en C

Lorsque nous demandons à Turbo C de compiler ce programme, il génère lui aussi des instructions en langage machine. Mais ici, toute référence à la variable i, par exemple, sera directement traduite (en son adresse), évitant ainsi une recherche lors de l'exécution. De même, la boucle sera également traduite, et le processeur saura retourner au test "i<=30000 ?" après avoir effectué l'incrémentation "i=i+1" sans avoir besoin de rechercher l'emplacement dans le programme de ce fameux test. La valeur finale est également testée lors de la compilation (est-elle du bon type ?). La majorité des tests nécessaires lors de l'exécution pour le programme en Basic sont effectués à la compilation.

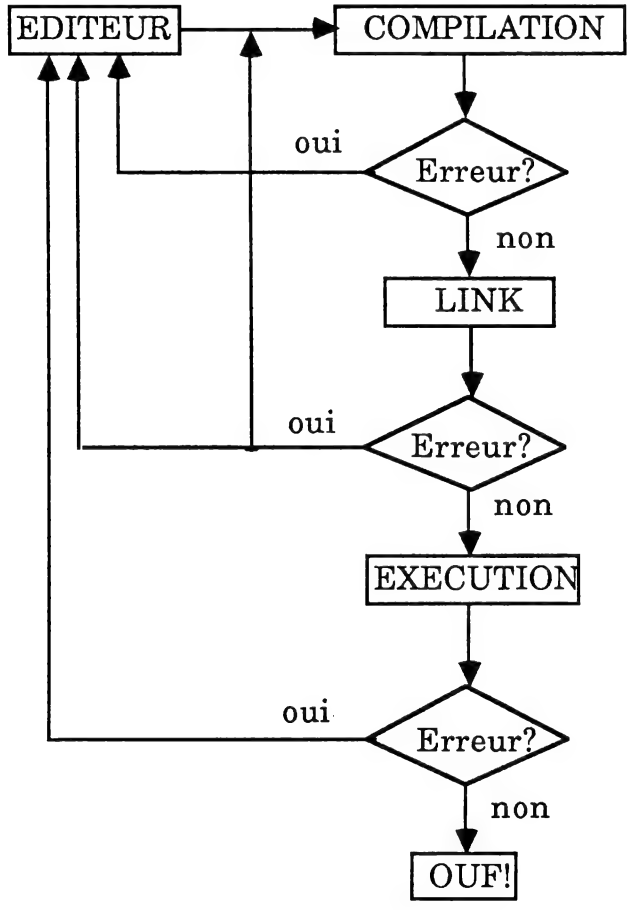
La grosse différence réside donc dans le moment où intervient la traduction en langage machine : en Basic interprété, elle est effectuée LORS DE L'EXECUTION et pour chaque instruction rencontrée, tandis qu'en C compilé elle intervient avant l'exécution, et en une seule fois. Cela procure de nombreux avantages : d'une part l'exécution est beaucoup plus rapide (moins de tests et de vérifications), d'autre part le programme est, une fois traduit, exécutable sans le compilateur, tandis qu'un programme interprété ne peut être exécuté que si l'interpréteur est présent en mémoire.

Quoi qu'il en soit, les interpréteurs reviennent à la mode car ils permettent une mise au point plus facile : si nous stoppons le programme Basic, nous pouvons vérifier facilement grâce à l'interpréteur quelle est la valeur de la variable I,(par exemple "PRINT I"). Lorsque nous arrêtons un programme compilé, nous retournons sous DOS et nous perdons d'un coup toute trace des variables et du programme. Il faut fouiller directement en mémoire octet par octet pour retrouver nos billes (avec un peu de chance et surtout de bons outils) .

D'autre part, la compilation prend un certain temps. Avec un interpréteur, le programme peut être exécuté directement après avoir été saisi. En C, la moindre modification dans le programme entraîne une recompilation de la totalité de celui-ci.

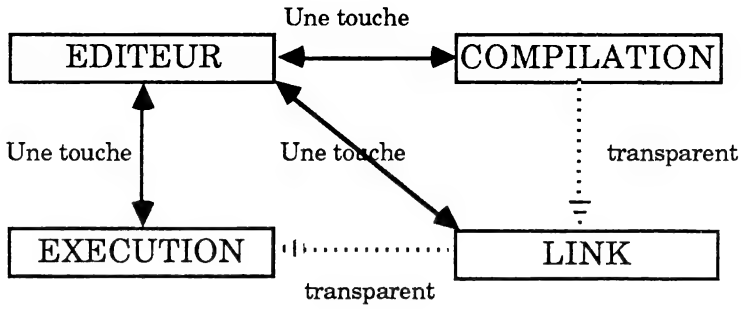
Enfin, autre avantage des interpréteurs, leur "intégration", si l'on peut dire. En effet, un interpréteur Basic par exemple, comporte généralement son propre éditeur. Il suffit de stopper l'exécution pour modifier le programme. Avec un langage compilé, il faut tout d'abord utiliser un éditeur de textes pour entrer le programme, puis lancer le Compilateur, puis le Linker, et enfin exécuter le programme obtenu. Si une erreur intervient, on arrête l'exécution, et on recommence : édition, compilation, link, exécution.

C Traditionnel



Toutefois, Turbo C diminue ce genre de manipulation, grâce à l'environnement intégré. En effet, Turbo C comporte un éditeur, un compilateur et un exécuteur. Le passage de l'un à l'autre se fait par le simple appui d'une touche. C'est le fameux effet "Turbo"!

TURBO C



TURBO C, LANGAGE TYPE, MODULAIRE, STRUCTURE (ET RUSE)!

Tout ceci ne rend pas C particulièrement original. Comme nous venons de le souligner, Pascal travaille de la même façon (il est compilé). Turbo-Pascal supprime même l'étape du Link!

Là où C apporte quelque chose d'original, c'est au niveau des "entités" évoquées plus haut. En effet, un programmeur dispose de trois éléments fondamentaux dans un langage : les variables -qu'il peut utiliser pour stocker des valeurs- les instructions ou fonctions -avec lesquelles il peut manipuler ces variables- et enfin un ensemble de règles, qui lui permettent de créer ou d'assembler ces deux types d'entités de façon à obtenir un programme cohérent et fonctionnel.

UN LANGAGE = TROIS ENTITES

→ ① VARIABLES

→ ② INSTRUCTIONS ET FONCTIONS

→ ③ REGLES D'ASSEMBLAGE DE 1 ET 2

Le C ne désorientera pas énormément les adeptes du Pascal dans un premier temps. En effet, comme ce dernier, C est basé sur les principes de la programmation structurée : un programme compliqué est découpé en tâches plus simples. Celles-ci sont elles-mêmes décomposées en sous-tâches, et ainsi de suite, jusqu'à obtenir un ensemble de tâches très simples à programmer et dont les interdépendances sont bien définies. On peut ensuite procéder à la programmation en sens inverse, niveau par niveau, de façon à ne programmer une routine que lorsque toutes ses sous-routines sont au point.

En C, il n'y a qu'un seul niveau de déclaration : une routine peut en appeler d'autres, mais ne peut pas en inclure elle-même une autre, à la différence du Pascal. Si l'on excepte cette subtilité (proche des subroutines du Fortran), un programme en C se conçoit à peu près de la même façon qu'en Pascal. C est un langage "modulaire" car une routine indépendante avec ses sous-routines constitue ce que l'on appelle un module.

C est, tout comme Pascal, un langage typé. Cela signifie que le langage procure non seulement des types de base pour les variables (nombre, caractères...), mais aussi que le programmeur pourra créer ses propres types si le besoin s'en fait sentir. Toutefois, à la différence cette fois du Pascal, C procure une similitude entre certains types de bases, et il en est de même pour les types créés. Les variables d'un type peuvent prendre des valeurs d'un autre type, si celui-ci est compatible avec le premier.

De plus, C est récursif de la même façon que Pascal : une routine peut s'appeler elle-même, soit directement, soit indirectement. Les variables sont empilées pour ne pas perdre leur trace à chaque nouvel appel.

Bien que C soit donc très proche de Pascal, il s'en distingue par ses nombreuses ruses. Il tient parfois du Basic (toute autre comparaison mise à part bien sûr) par les astuces qu'il autorise, astuces capables de rendre un programme aussi puissant et compact qu'illisible!

Les variables de C ont déjà des particularités en elles-mêmes : contrairement à ce qui est imposé dans les autres langages évolués, tous les types de variables sont compatibles entre eux, à peu de chose près. Cela signifie, par exemple, qu'un caractère (par exemple 'A') pourra être traité comme un nombre (65 en Ascii), ou une phrase comme une série de nombres.

De même, les instructions sont toutes basées sur un principe important : toutes les cohabitations imaginables sont possibles. On peut par exemple ranger une valeur dans une variable au cours d'une instruction de test, ou déclarer une nouvelle variable à l'intérieur d'une boucle, ou encore composer plusieurs instructions en une seule, etc...

Enfin, les règles d'assemblage de tous ces éléments sont elles-mêmes très tolérantes, conséquence et cause tout à la fois de toute la souplesse ainsi obtenue.

C, UN PASCAL AMELIORE ET DIMINUE?

Tout ceci est bien beau. Mais qu'en découle-t-il pour le programmeur Pascal, par exemple ? D'une façon générale (il y a toujours quelques petites exceptions), tout ce qui peut être fait en Pascal peut être programmé de façon pratiquement identique en C, surtout depuis la norme ANSI. En effet, les règles du C sont basées sur les mêmes idées théoriques que Pascal (la programmation structurée). Par contre, C a initialement été conçu pour des professionnels du développement et non pour des étudiants en informatique, et ne comporte donc ni les limitations pédagogiques, ni les "garde-fous" de Pascal. En Pascal, une erreur d'inattention conduit généralement à une erreur de compilation, et donc à une correction rapide. En C, on peut très bien obtenir un code exécutable qui fait quelque chose de tout à fait imprévu. Inversement, comme pratiquement tout est permis (y compris les erreurs!), on peut faire beaucoup plus de choses.

Enfin, mais c'est important, C a toujours été attaché à une notion de portabilité. En clair, cela signifie qu'un programme C ne doit à priori subir aucune modification sur le source (le fameux texte pour l'humain) pour effectuer le même travail de la même façon sur une machine ou un compilateur totalement différent. Cet aspect des choses a toujours A PEU PRES été respecté avec C; on ne peut pas en dire autant de tous les langages (Basic, portable?).

Cela signifie que, dès qu'un compilateur C est disponible sur une machine donnée, celle-ci peut disposer d'un grand nombre d'outils (provenant d'Unix, et donc programmés ... en C , puisque c'était l'objectif initial des concepteurs de ce langage!) et de logiciels. Ce qui est bien évidemment appréciable, et si C connaît un tel succès actuellement, c'est probablement en grande partie parce qu'il a eu la chance de ne pas donner naissance à d'innombrables dialectes incompatibles (il existe essentiellement trois familles de dialectes qui en fait ne se différencient pas énormément). La norme ANSI vient à point nommé pour figer (disons plutôt définir officiellement) une version de C résolument moderne et très puissante. Dans ces conditions, il est évident qu'aucun éditeur n'aura intérêt à s'éloigner trop de cette norme, car il risquerait de perdre un gros avantage et de s'isoler de l'univers riche du C.

TURBO C : ENCORE MIEUX QUE C...

La philosophie de C est en réalité assez particulière. C'est un langage à la fois évolué et de bas niveau : on peut utiliser des structures de données extrêmement complexes de façon simple, tout comme on peut utiliser des fonctions très simples d'une manière très complexe!

C'est probablement cette cohabitation de deux styles, l'un propre et net, l'autre efficace et rusé, qui fait le charme de C : il peut convenir pratiquement à tous les styles de programmation, et autorise le développeur à s'exprimer comme il le sent. S'il aime prendre des risques et utiliser à fond les possibilités cachées d'un langage, il trouvera son bonheur avec C. S'il préfère la philosophie Pascal, avec des listings très propres, très hiérarchisés, et misant essentiellement sur la

lisibilité ou la facilité de maintenance et d'évolution du logiciel, C ne le décevra tout de même pas. Et si enfin il aime mélanger propreté et efficacité, C sera encore adapté à ses besoins.

Le problème de toute cette souplesse, nous l'avons souligné à maintes reprises, c'est qu'un programme en C, même simple, peut comporter des vices de forme insoupçonnés. Comme tout est possible -ou presque- il est facile de se tromper, et de programmer quelque chose qui sera exécuté, mais qui ne correspond pas du tout à ce que l'on désirait obtenir. C'est vrai dans n'importe quel langage, mais le phénomène est très sensible en C. Les compilateurs C sont incapables de tenir compte de cela, puisque par définition même ils autorisent tout ce qui est risqué. Sous Unix, un utilitaire nommé LINT (il n'est jamais très loin du compilateur C!) permet de détecter, dans un programme, la majeure partie des situations ambiguës, ou susceptibles de ne pas correspondre aux désirs du programmeur (mais sans les interdire, bien entendu).

Turbo C possède l'avantage, lorsqu'on l'utilise en respectant la norme ANSI, de proposer également de nombreuses aides à la mise au point. Par exemple, il pourra indiquer quelles sont les causes possibles de mauvais fonctionnement, quelles variables n'ont apparemment pas été initialisées, etc.. Bien entendu, C étant très souple, le fait d'utiliser une variable non initialisée n'est pas forcément une erreur : c'est peut-être voulu. Le compilateur ne s'en préoccupe donc jamais. Mais Turbo C, si vous le désirez (on a parfois un doute), vous indiquera ce cas de figure. Il est important de savoir que Turbo C dispose d'une bonne partie des fonctions de LINT. Tous les compilateurs futurs s'en inspireront vraisemblablement car LINT provoque un effet d'accoutumance aussi certain que profitable!

Toutefois, l'utilisation de toutes les possibilités de la norme ANSI avec Turbo C, si elle donne des programmes compatibles avec les futurs compilateurs, peut dans certains cas nuire à la portabilité actuelle. N'oubliez pas que C existe depuis 15 ans, et que la plupart des compilateurs sont basés sur des standards assez particuliers. La norme ANSI sera sans aucun doute retenue par la suite, mais à l'heure actuelle, utiliser toutes ses possibilités revient à rendre les programmes dépendants de la norme. Pour cette raison, Turbo C, qui est astucieux, autorise à la fois la "nouvelle norme", l'ANSI, et l'ancienne, plus habituelle. Au programmeur de choisir!

D'autre part, Borland n'ayant pas failli à son habitude, Turbo C bénéficie avant tout de "l'effet Turbo". Toutes les opérations compliquées que nous avons décrites —compilations, link, détection des erreurs possibles et exécution— PEUVENT ETRE PROVOQUEES EN APPUYANT SUR UNE SEULE TOUCHE! Si une erreur survient, il suffit d'un appui de touche pour retrouver dans le source l'endroit de l'erreur. C'est quasiment magique! D'autre part, toutes les fonctions de Turbo C (et elles sont nombreuses) demandent au plus l'appui quatre touches successives. Un vrai régal.

Bien entendu, Turbo C est avant tout rapide. La vitesse de compilation (lorsque celle-ci est effectuée seule, par exemple seulement pour vérifier la syntaxe du programme) est véritablement stupéfiante; elle dépasse largement le record du Turbo Pascal, qui pourtant avait fait date. Les performances de Turbo C pour obtenir un programme exécutable (donc après Link, test des erreurs, et écriture sur disque) restent, malgré la quantité d'opérations complexes effectuées, supérieures à celles de Turbo Pascal (il est vrai que celui-ci, comme tout compilateur Pascal, vérifie beaucoup plus de choses, mais en revanche ne procède à aucun Link).

Turbo C peut être résumé en ces termes : il offre une version puissante de C, respectant la norme (avec les petits "plus" Borland, bien entendu), et un environnement de développement agréable, tant par ses performances pures que par les outils dont il dispose.

CHAPITRE 2

TURBO C SUR COMPATIBLES PC et AT

Contrairement à Turbo Pascal, Turbo C n'existe pour l'instant que sur IBM PC et compatibles. Il est probable qu'une version sur Macintosh paraîtra ultérieurement, mais certainement pas une version CP/M. En effet, le compilateur Turbo C est trop imposant (alors que Turbo Pascal 3.0 occupe à peine 40 Ko de mémoire, ce qui permettait de l'implanter sur des machines CP/M équipées de 64 Ko, comme l'Amstrad CPC).

Notez que Turbo C (en version environnement intégré) nécessite au minimum 384 Ko de mémoire. Si vous ne possédez qu'un seul lecteur de disquettes, sachez que le compilateur ne vous permettra pas vraiment de travailler sur de gros programmes. Les choses s'améliorent avec un deuxième lecteur mais l'idéal est bien entendu le disque dur.

Le système Turbo C est composé de quatre disquettes, ce qui fournit environ 1 méga-octet de fichiers variés. Inutile de dire que votre premier souci sera de copier ces disquettes pour ensuite les placer à l'abri. Même si vous possédez un disque dur, n'hésitez pas à effectuer une copie sur disquette supplémentaire. L'idéal est d'avoir une copie de travail, une copie de sauvegarde, ET les originaux dans un endroit sûr. Minimisez les risques de perturbations magnétiques. Ces trois copies doivent bien entendu être stockées dans des endroits différents (sinon, quel intérêt?).

LES FICHIERS DES DISQUETTES

Les quatre disquettes comportent un grand nombre de fichiers. La liste en est d'ailleurs assez inquiétante pour un débutant. Mais pas de panique : Borland a pensé à tout. La notice Turbo C explique comment installer Turbo C pour les trois configurations possibles (un lecteur, deux lecteurs, ou un disque dur). Nous vous recommandons de suivre ces conseils à la lettre. Il est possible que le compilateur fonctionne de façon erratique si vous ne suivez pas les directives de Borland.

Parmi les disquettes, vous allez trouver de nombreux fichiers comportant des extensions différentes. Outre les fichiers .EXE et .COM, qui sont des outils de TURBO C (TC.EXE et TCC.EXE sont les deux compilateurs), vous trouverez notamment des fichiers .H sur la disquette 3, et des fichiers .OBJ et .LIB sur les disquettes 3 et 4.

Les fichiers .H contiennent les déclarations nécessaires pour utiliser les fonctions de la librairie standard de Turbo C. Par exemple, la fonction PRINTF (équivalent du WRITE en Pascal et du PRINT en Basic) n'existe pas en C. Sa déclaration est située dans le fichier STDIO.H, et son code (ce qui permet de l'exécuter et donc de l'inclure dans le code du programme) se situe dans un fichier .LIB. Pour utiliser la fonction PRINTF dans un programme, il faudra donc inclure le fichier STDIO.H dans le source du programme. Nous verrons plus loin comment réaliser cette opération.

Les fichiers .LIB (librairies) contiennent le code machine de ces fonctions standard. Les librairies sont utilisées par petit bout lors de la compilation pour générer le code d'un programme, suivant les fonctions de la librairie qui ont été utilisées. Pour chaque utilisation de `printf` dans un programme, le compilateur copiera la portion du fichier LIB correspondant à cette fonction (la déclaration du fichier STDIO.H lui indiquant où trouver cette portion).

Enfin, les fichiers .OBJ sont utilisés lors du Link. Ils sont en quelque sorte des librairies, mais très spécialisées. Leur contenu est surtout lié au matériel et au modèle mémoire utilisé par la compilation. Nous reviendrons également sur ces notions ultérieurement.

INSTALLATION DE TURBO C

Tous les fichiers ne sont pas absolument nécessaires pour utiliser Turbo C. Bien entendu, l'installation de Turbo C dépendra de votre configuration. Examinons brièvement les différents cas.

Un seul lecteur de disquette

Si vous ne possédez qu'un seul lecteur, vous devrez préparer deux disquettes. La disquette PROGRAMME, et la disquette de travail.

Sur la disquette programme, copiez les fichiers TC.EXE et TCHELP.HLP, qui se situent sur la disquette 1 de Turbo C.

Sur la disquette de travail, il faut tout d'abord créer deux répertoires : C_LIB et C_INCL. Dans le répertoire C_INCL, vous copierez les fichiers .H du disque 3 (n'oubliez pas STAT.H qui se trouve dans le répertoire SYS de ce même disque 3). Dans le répertoire C_LIB, vous copierez les fichiers suivants :

C0S.OBJ

EMU.LIB

FP87.LIB

MATHS.LIB

CS.LIB

Tous ces fichiers se trouvent sur la disquette n° 3. Pour l'instant, il n'est pas important de savoir à quoi servent ces cinq fichiers. Sachez seulement qu'ils sont tous indispensables pour générer un programme exécutable.

Sur la disquette de travail, vous placerez également vos propres programmes.

En résumé, voici donc vos deux disquettes :

Disque 1 : PROGRAMME.

A:\TC.EXE A:\TCHELP.TCH

Disque 2 : TRAVAIL.

A:\programmes.C A:\C_LIB\C0S.OBJ A:\C_LIB\EMU.LIB A:\C_LIB\FP87.LIB A:\C_LIB\MATHS.LIB A:\C_LIB\CS.LIB A:\C_INCL\fichiers.H

Pour lancer TURBO C, vous devez d'abord placer le disque PROGRAMME dans le lecteur et taper TC suivi de ENTER. Ensuite, une fois l'écran affiché, vous pouvez placer votre disquette de travail dans le lecteur.

Deux lecteurs

La démarche à suivre est la même que pour les systèmes à un seul lecteur, mais vous pouvez alors placer la disquette programme en A: et la disquette travail en B: Veillez alors à vous trouver en A: pour lancer TC.EXE.

Un disque dur

Le travail sur disque dur est plus agréable. Nous vous recommandons de créer tout d'abord les répertoires suivants :

C:\C C:\C\C_LIB C:\C\C_INCL C:\C\PROGS

Le répertoire \C reçoit alors les programmes de Turbo C soit les fichiers .EXE et .COM des disques 1 et 2, \C\C_LIB reçoit les fichiers .LIB et C0x.OBJ des disques 3 et 4, \C\C_INCL reçoit les fichiers .H du disque 3. Le répertoire \C\PROGS recevra pour sa part vos programmes C.

CONFIGURER TURBO C

Quelle que soit votre configuration, vous devez indiquer à Turbo C où se situent les librairies et les fichiers H. Pour cela, vous devez utiliser d'une part TCINST, d'autre part une option de Turbo C. Voici la démarche à suivre pour chaque configuration (en supposant que vous ayez suivi nos conseils).

Un seul lecteur

- Placez la disquette PROGRAMME dans le lecteur.

- Tapez "TC" suivi de ENTER.

- Tapez la touche ALT et, sans la relâcher, la touche O. Cela vous place dans le menu OPTIONS.

- Tapez "E" pour passer dans le sous-menu Environnement.

- Tapez "I", puis "C_INCL" suivi de ENTER.

- Tapez "L", puis "C_LIB" suivi de ENTER.

- Tapez la touche "ESCAPE" pour remonter au menu OPTIONS.

- Tapez "S" pour sauver ces options. Eventuellement, répondez "Y" à la demande de confirmation si le fichier TCCONFIG.TC existe déja.

- Vous pouvez maintenant charger la disquette TRAVAIL.

- Utilisez de nouveau la touche "S" dans le menu OPTIONS pour sauvegarder les options sur la disquette de travail.

- Tapez "ALT" et "X" pour sortir de Turbo C.

Ouf! Turbo C est installé.

Deux lecteurs

- Placez la disquette Turbo C n°1 dans le lecteur A.

- Tapez "A:" si vous êtes sur le lecteur B.

- Tapez "TCINST" suivi de ENTER.

- Placez la disquette PROGRAMME en A:, et la disquette TRAVAIL en B:

- Choisissez l'option "Turbo C Directory" en tapant "T".

- Tapez "A:\" suivi de Enter.

-Tapez succesivement "Q" et "Y" pour quitter TCINST et sauver l'installation dans TC.EXE. Vous revenez ainsi au DOS.

- Tapez "TC" suivi de ENTER.

- Tapez la touche ALT et, sans la relâcher, la touche "O". Cela vous place dans le menu "Options".

- Tapez "E" pour passer dans le sous-menu "Environnement".

- Tapez "I", puis "B:\C_INCL" suivi de ENTER

- Tapez "L", puis "B:\C_LIB" suivi de ENTER

- Tapez la touche "ESCAPE" pour sortir du sous-menu.

- Tapez "S" pour sauver les options. Eventuellement, répondez "Y" à la demande de confirmation si le fichier TCCONFIG.TC existe déjà.

- Tapez "ALT" et "X" pour sortir de Turbo C.

Turbo C est installé ! Pour travailler, placez vous en B: et lancez Turbo C par la commande "A:TC".

Un disque dur

- Tapez "C:" suivi de ENTER si vous êtes sur le lecteur A.

- Tapez la commande suivante (vous pouvez même l'ajouter dans AUTOEXEC.BAT pour éviter de la retaper à chaque mise en route de l'ordinateur) : PATH C:\C.

- Tapez "CD C".

- Tapez "TCINST" suivi de ENTER

- Choisissez l'option "Turbo C Directory" en tapant "T".

- Tapez "C:\C" suivi de ENTER.

-Tapez successivement "Q" et "Y" pour quitter TCINST, sauver l'installation dans TC.EXE et revenir au DOS.

- Tapez "TC" suivi de ENTER.

- Tapez la touche ALT et, sans la relacher, la touche "O". Ceci vous place dans le menu "Options".

- Tapez "E" pour passer dans le sous-menu "Environnement".

- Tapez "I", puis "C:\C\C_INCL" suivi de ENTER.

- Tapez "L", puis "C:\C\C_LIB" suivi de ENTER.

- Tapez la touche "ESCAPE" pour sortir du sous-menu.

- Tapez "S" pour sauver les options. Eventuellement, répondez "Y" à la demande de confirmation si le fichier TCCONFIG.TC existe déjà.

- Tapez "ALT" et "X" pour sortir de Turbo C.

Turbo C est installé. Pour travailler, placez vous dans le répertoire C:\C\PROGS.

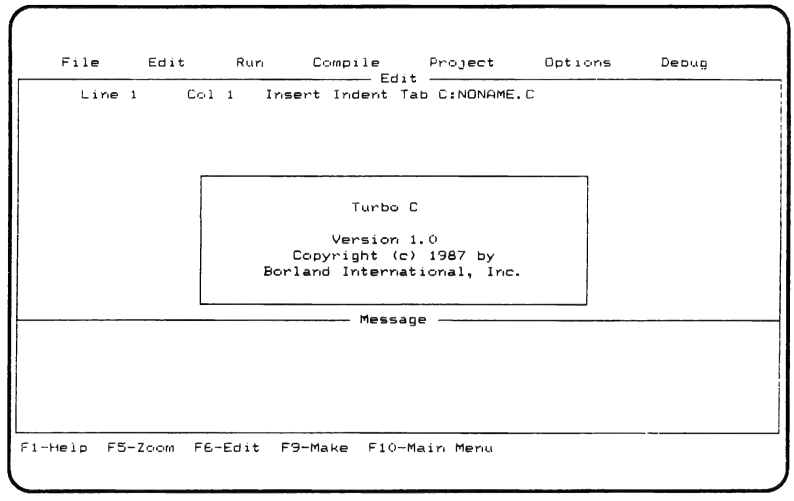
Le chemin d'accès sur disque

Bien entendu, cette installation est définitive. Il n'est pas nécessaire de procéder à ces manipulations avant chaque séance de travail, si vous avez suivi nos conseils. Les utilisateurs de disque dur auront avantage à placer une commande "PATH C:\C" dans le fichier AUTOEXEC.BAT du répertoire C:\, sans quoi ils ne pourront accéder à TC.EXE qu'à partir du répertoire C:\C, ce qui les obligera à changer sans arrêt de répertoire pour atteindre les programmes situés dans C:\C\PROGS. Si vous possédez déjà une commande PATH dans votre AUTOEXEC, il suffit d'ajouter votre nouveau chemin à la suite des précédents, séparé par un point-virgule. Par exemple :

```
PATH C:\; C:\UTIL; C:\BATCHS; C:\C;
```

L'ENVIRONNEMENT INTEGRE DE TURBO C

Si tout s'est bien passé, vous voilà prêts à apprendre Turbo C. Tapez donc "TC". Si le système vous renvoie un message d'erreur, c'est que vous n'êtes pas dans le répertoire où se situent TC.EXE, TCHELP.TCH et TCCONFIG.TC. Nous vous conseillons alors de reprendre les manipulations décrites ci-dessus, car vous avez dû commettre une erreur quelque part. Sinon, vous vous retrouvez devant l'écran suivant :



Bienvenue sous Turbo C

Vous pouvez constater que l'écran comporte trois parties. En haut, une ligne qui indique les menus de commandes disponibles. Nous l'appellerons "barre des menus". En dessous, une fenêtre EDIT, et enfin une fenêtre MESSAGE.

Examinons tout d'abord la barre des menus. Vous observez un certain nombre de commandes sur cette ligne. En fait, vous avez accès à tout moment aux menus et fonctions de cette ligne en tapant la touche "ALT" puis, sans relâcher celle-ci, l'initiale de la fonction voulue. Voici l'effet des touches de commande :

ALT + F = "File" : affiche le menu "FILE", pour diverses opérations sur les fichiers de vos programmes. Ce menu est assez explicite en lui-même.

ALT + E = "Edit" : passe directement dans la fenêtre "EDIT". Vous pouvez ensuite taper ou modifier le programme situé dans cette fenêtre.

ALT + R = "Run" : provoque l'exécution du programme situé dans la fenêtre d'édition (sauf si la gestion de projet est en fonction).

ALT + C = "Compile" : affiche un menu pour les différentes étapes de compilation. Ce menu nous sera inutile. Il sert à générer simplement un fichier OBJET pour le linker avec un autre, et à lancer les utilitaires de gestion de projet (voir ci-dessous). Il est aussi utilisé sur de gros programmes pour tester la syntaxe avant de lancer un link parfois long.

ALT + P = "Projet" : affiche un menu pour la gestion de projet. Tout comme le menu COMPILE, nous n'utiliserons pas ce menu tout de suite. Il est destiné à faciliter le travail sur de gros programmes constitués de plusieurs fichiers source.

ALT + O = "Options" : Nous avons déjà utilisé cette commande puisqu'elle nous a permis d'installer et de configurer TURBO C. Il s'agit d'un menu OPTIONS. Il n'est pas très utile à notre stade. La plupart des options proposées sont destinées aux programmeurs accomplis !

ALT + D : le menu "DEBUG" comporte quelques options dont nous verrons l'utilité, mais il affiche aussi la taille mémoire disponible à tout moment.

En pratique, seules serviront, pour l'instant, les commandes ALT+E pour taper un programme, ALT+F pour le sauver ou le recharger, et ALT+R pour l'exécuter. Par conséquent, il ne faut surtout pas s'inquiéter du nombre d'options proposées par Turbo C : elles sont, comme l'indique clairement leur nom, optionnelles. Tel quel, Turbo C vous permet de toute façon de travailler.

La deuxième partie est la fenêtre EDIT. Dans cette fenêtre se trouve le source du programme sur lequel nous travaillons. Pour l'instant, elle est vide et indique que le fichier traité est "NONAME.C", ce qui signifie en bon Français "SANS NOM". Les mentions "Insert", "Indent" et "Tab" indiquent respectivement :

- si la frappe se fait en insertion ("Insert" affiché), ou en recouvrement (par dessus les caractères situées à l'endroit du curseur) si "Insert" n'est pas affiché.

- si l'éditeur est en indentation automatique ("Indent" affiché) : le passage à la ligne place le curseur à la verticale du premier caractère de la ligne précédente ou non automatique (le passage à la ligne place le curseur au début de la ligne suivante).

- si le mode tabulation est standard ("Tab" est affiché). Dans ce mode, la touche de tabulation déplace le curseur de 8 espaces. Dans le mode automatique ("tab" n'est plus affiché), le curseur est envoyé à la verticale du mot suivant situé sur la ligne précédente. C'est une option pratique pour aligner les mots.

La dernière partie de l'écran est la fenêre MESSAGE. Turbo C y inscrit ses conclusions lors des compilations. Nous verrons l'interêt de cette fenêtre dans quelques pages !

AU TRAVAIL!

Pour prendre la mesure du système Turbo C, nous allons immédiatement réaliser notre premier programme (nous savons que vous attendez ce moment avec impatience ...).

Nous aimerions vraiment taper dans l'originalité, pour ce premier programme, mais C est un de ces langages qui ont des traditions. Justement, la tradition veut que le premier programme d'un néophyte en C se nomme HELLO.C, et soit le suivant :

```
main()
{
   printf("hello, world !\n");
}
```

Et en plus, c'est en anglais... Ne nous en veuillez pas : ce programme est le premier programme exemple du livre de D. Ritchie (qui est, rappelons le, le créateur d'Unix et du langage C). Et ce livre, assez ancien et largement critiqué tant en bien qu'en mal, est tout de même la "bible" du langage C, puisque c'est son créateur qui l'a écrit.

Cet exemple de programme, HELLO, aussi inutile que futile est devenu, par plaisanterie puis par habitude, une tradition; tout comme le PRINT "COUCOU" en Basic ou le PROGRAM TOTO en Pascal. Vous remarquerez d'ailleurs que Borland l'a placé sur la disquette 1 de Turbo C. Tradition oblige !

Voyons donc comment programmer notre HELLO.C !

Tout d'abord, il faut indiquer à Turbo C que nous désirons travailler sur le source HELLO.C

Pour cela, nous allons utiliser ALT+F. Le menu des fichiers s'affiche. Il faut ensuite soit déplacer le curseur avec les flèches sur la commande "Load" et taper ENTER, soit, ce qui est plus rapide, taper l'initiale de la commande, soit "L".

Nous vous conseillons d'utiliser les initiales des commandes plutôt que le curseur et les flèches suivies de Enter. Au bout de quelques jours de travail, vous vous souviendrez facilement des successions de touches utiles, et vous gagnerez un temps fou. Mais vous êtes bien entendu libres de choisir une autre façon de travailler : Turbo C vous permet d'accéder aux menus de deux ou trois façons différentes !

Par exemple, si vous n'aimez pas la touche ALT (pourquoi pas ?), sachez que la touche de fonction F10, à tout moment, vous replacera sur la barre des menus, d'où vous pouvez appeler les menus grâce au curseur ou aux initiales.

Maintenant que nous avons validé la commande "LOAD", Turbo C nous demande, à sa manière, un nom de fichier. En ce qui nous concerne, ce sera HELLO.C et nous tapons donc ce nom, suivi de ENTER (si vous validez sur un nom "blanc", Turbo C affiche la liste des fichiers accessibles et un curseur permet de s'y déplacer pour en choisir un).

Le menu "FILE" est toujours présent, mais notre "HELLO.C" est venu s'inscrire sur la première ligne de la fenêtre EDIT, remplaçant le fort peu sympathique "NONAME.C" qui s'y trouvait. Pour passer en édition, il suffit d'utiliser ALT+E. Cela nous envoie directement dans la fenêtre EDIT, ligne 1, colonne 1, et nous pouvons taper le programme suivant :

```
main()
{
   printf("hello, world !\n")
}
```

Ah, êtes-vous assez attentif ? Regardez bien : nous avons enlevé un point-virgule ! Ne vous inquiétez pas, nous l'avons fait exprès. En fait, c'est une erreur de syntaxe. Vous allez tout de suite comprendre pourquoi nous avons pris la liberté de provoquer une telle erreur.

Notez que les symboles '{' et '}', très importants en C, sont l'équivalent du BEGIN et END du Pascal. Ils indiquent le début et la fin de bloc d'instructions. Si vous possédez un clavier QWERTY, tout va bien. Si par malheur (!) c'est un clavier AZERTY, il vous faudra probablement jongler avec les touches CTRL et ALT pour obtenir ces caractères.Voici les combinaisons de touches les plus fréquentes. Vous trouverez bien celle qui fonctionne ! Notez que la méthode "ALT+nombre" fonctionne dans tous les cas, mais malheureusement c'est la moins agréable. Sur les claviers récents, il suffit de taper "ALT", la touche CTRL n'est plus nécessaire. Notez que ALT et CTRL doivent être maintenues pendant que l'on tape sur les autres touches, puis relâchées.

{ : CTRL + ALT + "("

ALT + "("

SHIFT + CTRL + ALT + "^" (à côté de "P")

ALT + "123" tapé au clavier numérique (NUM LOCK enfoncé ou non)

} : CTRL + ALT + ")"

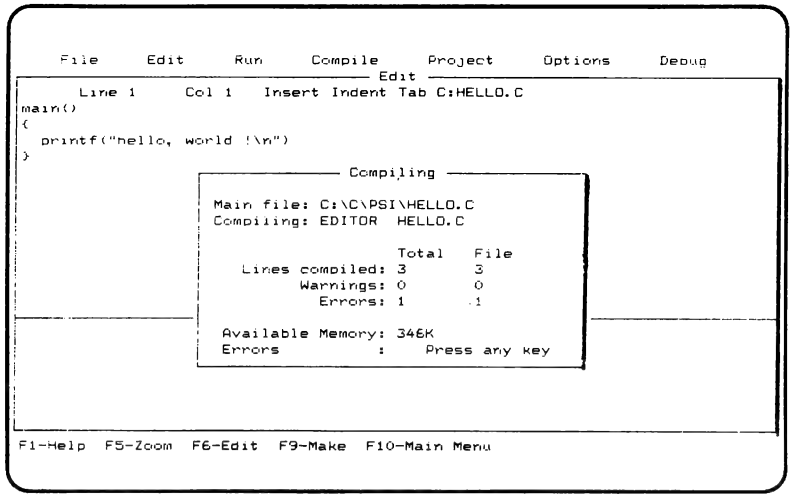
ALT + ")"

SHIFT + CTRL + ALT + "$" (à côté de "^")

ALT + "125" tapé au clavier numérique (NUM LOCK appuyé ou non)

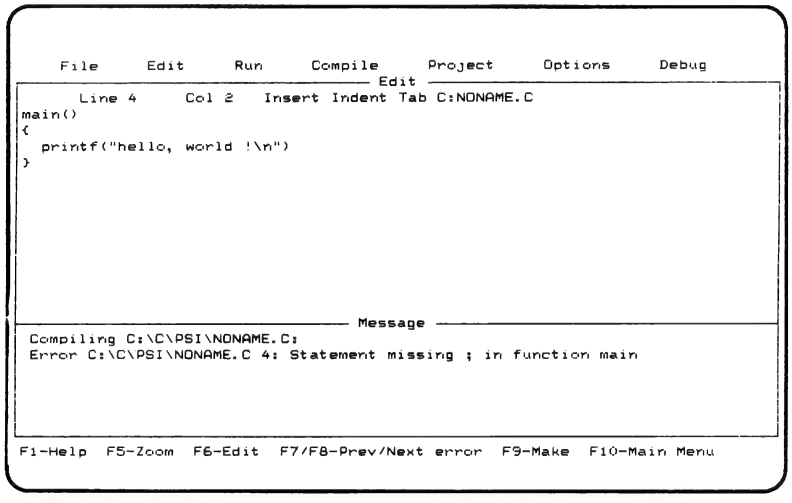
Vous avez probablement fini de taper ce programme erratique. Maintenant, il faut le compiler. Etant donné sa petite taille, nous allons même l'exécuter directement. Pour ce faire, il nous suffit de taper ALT + R, ce qui lance la commande RUN (Turbo C est le seul compilateur C actuel proposant cette commande). Vous remarquerez que Turbo C enchaîne, sur cette commande, la compilation, le Link et l'exécution sans autre intervention. Quelle facilité !

Mais voilà que la compilation nous envoie immédiatement un message erreur :



Une erreur a stoppé la compilation

Turbo C indique qu'il a détecté une erreur en compilant le source HELLO.C provenant de EDITOR (c'est-à-dire de notre fenêtre édition). Appuyons sur une touche, comme il nous le demande. L'écran s'enrichit alors de nouvelles informations :



Les erreurs sont affichées dans la fenêtre MESSAGE

La fenêtre message résume ce qu'a fait Turbo C. Il a lancé la compilation, ce qui donne le MESSAGE "Compiling etc...". Puis il a détecté un erreur :

```
Error C:\C\PROGS\HELLO.C 4: Statement missing ; in function mai
```

Ce message est mis en évidence (inversion vidéo sur un moniteur monochrome, couleur différente sur un moniteur couleur), ainsi que la ligne qui s'y rapporte dans la fenêtre EDIT (celle contenant "}").

Le message signifie qu'il manque un ";" dans une instruction de la fonction "main". C'est tout à fait exact, puisque nous l'avons enlevé ! Si la ligne mise en évidence n'est pas la bonne mais la suivante, c'est parce que le compilateur ne sait qu'il manque le point-virgule qu'en atteignant l'instruction suivante, ou un autre symbole indiquant la fin du `printf`. En l'occurrence, il cherche un point virgule, mais rencontre "}" avant de l'avoir trouvé. Le point-virgule pourrait parfaitement se trouver sur la ligne suivant le `printf`, et non sur la même ligne. Il faut seulement qu'il soit présent avant "}", un point c'est tout. C'est pourquoi le compilateur s'arrête sur le "}".

Passons en éditeur : ALT + E (vous devez commencer à en prendre l'habitude). Vous remarquez alors que Turbo C nous place sur la ligne contenant l'accolade fermée. En utilisant la flèche curseur "HAUT" (touche "8" du pavé numérique), puis la touche "Fin" ou "End" (pavé numérique), nous nous positionnons au bout de la ligne contenant le `printf`. Il nous suffit de rajouter le fameux point-virgule. Le programme est maintenant correct.

Pour le vérifier, il suffit de recompiler : ALT + R. Et voilà, à nos yeux éblouis surgit le résultat : l'ordinateur nous dit bonjour en anglais. Nous avons réalisé notre premier programme en Turbo C (et en plus nous avons respecté la tradition !).

Maintenant, sortons de Turbo C. Pour cela, le plus simple est d'utiliser la combinaison de touche ALT + X, qui fonctionne dans tous les cas de figure (mais vous pouvez aussi invoquer le menu FILE par Alt+F et demander l'option QUIT). De retour au DOS, effectuons la commande DIR :

```
C:\C\PSI>dir hello

 Volume in drive C is ASYNCHRONE
 Directory of  C:\C\PSI

HELLO     C          43   10-23-87    1:24p
HELLO     OBJ       218   10-23-87    1:24p
HELLO     EXE      5400   10-23-87    1:24p
        3 File(s)    2813952 bytes free

C:\C\PSI>
```

Notre programme occupe 5400 octets

Le fichier HELLO.C est notre programme. HELLO.OBJ est le fichier généré par le compilateur proprement dit. Il ne nous intéresse pas réellement car il n'est pas exécutable. Par contre. HELLO.EXE est le fichier exécutable. Vous pouvez d'ailleurs le constater en demandant :

```
C>HELLO

hello, world!

C>_
```

Le programme que nous avons exécuté tout à l'heure est donc directement utilisable sous MSDOS, sans passer par Turbo C. Toutefois, vous vous demandez peut-être pourquoi il occupe 5 400 octets, quand le source n'en demande que 41 et le programme compilé 217 ?

La raison de cet encombrement est la présence de la fonction `printf`: c'est elle qui est en grande partie responsable de cet accroissement de taille. Rassurez-vous, la proportion d'encombrement supplémentaire due aux librairies diminue au fur et à mesure que vos programmes grossissent.

Nous allons d'ailleurs modifier notre programme HELLO. Repassons sous TC, et rechargeons le programme HELLO. Puis utilisons ALT+E pour passer en éditeur, et changeons "`printf`" en "`puts`". Le programme devient :

```
main()
{
   puts("hello, world !\n");
}
```

L'exécution par ALT+R provoque le même résultat, car "`puts`" est une restriction de "`printf`" à l'écran. La fonction `printf` a en fait beaucoup plus de possibilités car elle peut par exemple travailler sur des fichiers (nous y reviendrons).Quittons Turbo C en faisant ALT+X, et redemandons DIR. Nous obtenons l'écran figuré sur la page suivante.

Il y a un progrès : en changeant simplement la fonction d'affichage, et bien que nous obtenions le même résultat, nous avons gagné environ 1 600 octets, soit presque 30 % de la taille.

Notez toutefois que le même programme, rédigé en langage machine pur, occuperait à peine 30 octets. Mais C n'est pas destiné à programmer des HELLO.C, il faut en convenir.

```
C:\C\PSI)dir hello

 Volume in drive C is ASYNCHRONE
 Directory of  C:\C\PSI

HELLO     BAK         43   10-23-87    1:24p
HELLO     C           41   10-23-87    1:28p
HELLO     OBJ        218   10-23-87    1:28p
HELLO     EXE       3808   10-23-87    1:28p
        4 File(s)    2813952 bytes free

C:\C\PSI)hello
hello, world !


C:\C\PSI)
```

Puts occupe environ 1600 octes de moins que printf

RESUMONS NOUS

Pour l'instant, nous avons réalisé un programme HELLO tout à fait simple. Nous avons vu que ALT+E passait en édition, que ALT+R exécutait un programme. Nous avons également utilisé la fenêtre message pour corriger une erreur, et le menu FILE pour sauver notre programme.

Avec ces éléments, vous en savez suffisamment pour commencer la programmation en Turbo C, qui sera abordée dans le chapitre suivant.

L'EDITEUR

Avant cela toutefois, un dernier mot sur l'éditeur :

L'éditeur de Turbo C est du type plein écran. Avec les flèches de curseur (sur le pavé numérique), vous pouvez déplacer celui-ci à travers tout le texte. Les touches spéciales de curseur (Home, End, PgUp et PgDn) ont également une fonction :

- La touche HOME ramène le curseur au début de la ligne.

- La touche END l'envoie après le dernier caractère

- PGUP et PGDN font respectivement monter et descendre le curseur d'une page de texte.

- INS quant à elle fait alterner entre les modes "INSERTION" et "RECOUVREMENT". En mode insertion, les caractères tapés repoussent les caractères qui les suivent. En mode recouvrement, ils les "écrasent".

- DEL efface le caractère situé à l'endroit du curseur et ramène le reste du texte.

- BACKSPACE efface le caractère qui précède le curseur puis recule le texte à partir de celui-ci.

L'éditeur comporte également un grand nombre de commandes diverses. Nous avons retenu les plus couramment utilisées ci-dessous. Pour les autres, nous n'avons qu'un seul conseil à vous donner : essayez-les toutes (reportez-vous au manuel), et forcez vous au début à les utiliser, dès que vous aurez un peu de temps. Vous finirez par trouver celles qui vous sont réellement utiles.

CTRL-PgUp : retour au tout début du texte (premier caractère).

CTRL-PgDn : saut à la fin du texte (dernier caractère).

CTRL- <- : saut au mot précédent.

CTRL- -> : saut au mot suivant.

CTRL-Y : supprime une ligne.

CTRL-Q puis Y : supprime la fin de la ligne à partir du curseur.

CTRL-Q puis L : annule les dernières modifications apportées à une ligne, pourvu qu'on se trouve toujours sur celle-ci et que l'on ne l'ait pas quittée entre temps.

CTRL-Q puis F : permet de rechercher un mot ou une phrase dans le texte. Il vous est demandé la chaîne à rechercher, puis une "option". Il existe beaucoup d'options, la plus utilisée étant "u", qui permet d'effectuer une recherche sans tenir compte des différences entre majuscules et minuscules.

CTRL-Q puis A : permet de rechercher et remplacer un mot par un autre dans le texte. Les options utiles sont "u" pour ignorer la différence entre majuscule et minuscule, "g" pour effectuer la recherche sur tout le texte (et non seulement la première occurrence à partir du curseur), et "n" si l'on désire que la substitution soit effectuée sans demande de confirmation (attention, cette dernière option peut être dangereuse si vous demandez l'option "gun", vérifiez que vous ne modifiez que les mots voulus). Vous pouvez combiner ces options entre elles.

CHAPITRE 3

PREMIERS PAS EN TURBO C

Dans ce chapitre, nous allons examiner les bases du langage C, sans trop entrer dans le détail. Vous devez avant tout vous installer tranquillement et confortablement devant votre écran, Turbo C chargé et sous vos ordres. Une fois ceci fait, vous voici prêts pour une première exploration !

QU'EST-CE QU'UN PROGRAMME EN TURBO C?

Vous vous souvenez sans doute de notre HELLO.C ? Le revoici, avec son point-virgule cette fois, car il va nous permettre de faire plus ample connaissance avec le langage C.

```
main()
{
  printf("hello, world !\n");
}
```

Dans ce minuscule programme, nous pouvons constater plusieurs choses. La première, c'est la présence d'une ligne au début du programme, qui comporte le mot "`main`" et deux parenthèses vides. Puis, suit une instruction "`printf`" avec une phrase, cette instruction étant placée entre accolades.

Ce qui est placé entre accolades se nomme BLOC EXECUTOIRE. Il s'agit d'un bloc qui, comme son nom l'indique, comporte des INSTRUCTIONS EXECUTABLES. Nous verrons plus tard qu'un bloc peut contenir d'autres sous-blocs, qui peuvent eux-mêmes en contenir d'autres et ainsi de suite. Il vous suffit de retenir qu'en langage C, un bloc a la structure suivante :

```
Accolade ouverte
  Déclaration de variables locales
  Instructions
Accolade fermée
```

(Nous verrons un peu plus loin ce qu'est une variable locale)

Remarque importante

Toutes les instructions, en C, doivent être terminées par un point-virgule. Ce symbole représente en effet le "terminateur". Son absence provoque une erreur de compilation, comme nous l'avons d'ailleurs constaté dans le chapitre précédent. Par contre, aucun point-virgule ne doit suivre la déclaration d'une fonction, ni une accolade fermée. LE POINT-VIRGULE TERMINE UNE

INSTRUCTION ET RIEN D'AUTRE, MAIS IL EST OBLIGATOIRE !!! Il y a fort à parier que vous en oublierez quelques-uns lors de vos débuts (et même plus tard !), ou que vous en placerez à des endroits interdits par inadvertance. Le compilateur vous indiquera toujours une erreur dans ce cas. Ayez toujours le réflexe de regarder avant tout les point-virgules (y en a-t-il en trop, les instructions en possèdent-elles toutes un ? etc...).

Quand un tel bloc n'est contenu dans aucun autre, il constitue une sorte de "super bloc". Ce super bloc se nomme FONCTION. Pourquoi ? Parce que, nous le verrons, nous pouvons passer des paramètres à ce bloc, et nous arranger pour qu'il puisse également renvoyer un résultat à la portion de programme qui l'appelle. C'est donc bien une fonction. Nous verrons aussi que, paradoxalement, une fonction en langage C peut fort bien ne rien recevoir du tout, et/ou ne rien renvoyer non plus (cette sorte d'exploit est totalement inutile d'un strict point de vue mathématique, mais nous verrons pourquoi que cela sert souvent C !).

En langage C, un programme n'est rien d'autre qu'un ensemble d'une ou plusieurs fonctions qui peuvent s'appeler les unes les autres. Toutes les fonctions d'un programme ont une même "importance", c'est-à-dire qu'elles sont toutes traitées de la même façon; mais l'une d'entre elles a néanmoins un rôle particulier. Son nom est imposé, il s'agit (vous l'avez peut-être deviné) de la fonction main (qui signifie principal en anglais). Elle indique au langage C qu'elle constitue le corps du programme proprement dit.

L'exécution d'un programme C débute toujours à la fonction "main", et celle-ci est obligatoire.

Tout le programme peut bien sûr éventuellement être contenu dans la fonction principale "main". C'est par exemple le cas pour le programme HELLO, qui ne comporte qu'une seule instruction. Mais dès qu'un programme comporte une tâche qui pourrait être résumée en une instruction ou fonction, il convient de créer une fonction C (avec un nom adéquat). C'est le but de ce qu'on appelle la structuration. Dans l'idéal, la taille d'une fonction ne devrait jamais dépasser deux ou trois pages écran. Si c'est le cas, il y a certainement dans cette fonction une portion de code qui peut être transformée en une autre fonction. Nous reviendrons plus en détail sur la programmation structurée.

Nous avons un dernier mystère à éclaircir dans HELLO.C : pourquoi y a-t-il deux parenthèses vides derrière la déclaration de la fonction main ? La raison en est simple. Comme nous l'avons laissé entendre, toute fonction C peut recevoir des valeurs. Ces valeurs, que nous appellerons bientôt paramètres, sont placées justement dans les parenthèses qui suivent le nom de la fonction. Lorsqu'aucun paramètre n'est nécessaire, il faut néanmoins laisser les parenthèses vides, justement pour indiquer au compilateur Turbo C que la fonction ne possède pas de paramètre ! C'est le cas de notre programme, qui ne reçoit bien aucun paramètre.

RESUMONS NOUS

En résumé, un programme est composé d'un ensemble de fonctions dont l'une est la fonction main. Une fonction comporte une déclaration et un bloc exécutoire. Le bloc exécutoire comporte des déclarations de variables et des instructions, lesquelles peuvent inclure d'autres blocs exécutoires. En fait, c'est donc assez simple. Il suffit de ne pas s'embrouiller !

PREMIERES DONNEES EN C

Le langage C ne serait pas très intéressant s'il ne comportait pas la possibilité de gérer des variables. Pour parler de variables, nous devons avant tout parler de types de données.

Toutes les données manipulées par les instructions du langage C ont un type. C comporte essentiellement trois types importants que l'on retrouve pratiquement dans tous les langages, par exemple Basic et Pascal.

Le type `int` concerne les nombres entiers. Il correspond très exactement, sous la forme `int`, au suffixe "%" des Basic Microsoft, et au type `integer` du Pascal. Toutefois, C possède plusieurs types d'entiers, comme nous le verrons par la suite.

Le type `int` permet de représenter les nombres entiers -32768 à +32767. Comme vous l'avez peut-être deviné, ceci vient du stockage de ces nombres, qui s'effectue sur 16 bits (2 octets) dont un est réservé au signe. Mais il n'est pas forcément utile de retenir ces détails. Retenez bien par contre les limites de ces nombres, et ayez-les à l'esprit constamment lorsque votre programme affichera un comportement suspect. Car la logique des calculs entiers fait que 32767+1 devient mystérieusement -32768 ! En fait ce n'est aucunement mystérieux, mais il faut le savoir. De même, si vous enlevez 1 à -32768, vous allez obtenir 32767. Ce ne correspond pas tout à fait avec ce qu'on apprend à l'école... Donc prudence et méfiance lorsque vous travaillerez avec de grands nombres !

Le second type utile est `float`. Celui-ci représente les nombres réels (comportant une partie décimale) et est l'équivalent du suffixe "!" des Basic Microsoft (valeurs réelles ensimple précision) et du type REAL en Pascal. Le plus petit réel proche de zéro est 3.4E-38 (ou -3.4E-38), le plus grand réel est 3.4E+38 (ou -3.4E+38). Ce sont, tout comme pour le format `int`, les limites classiques de ce type de données. Si vous vous demandez pourquoi les nombres réels se nomment "flottants" ou "à virgule flottante", nous en reparlerons tout à l'heure lorsque nous nous pencherons plus précisément sur leur cas.

Enfin, le type `char` représente les caractères. Il comprend les caractères ASCII (codes 0 à 127) ainsi que ceux ayant les codes 128 à 255, donc le jeu complet de caractères IBM-PC (y compris les caractères graphiques et spéciaux).

Il y a donc trois types de base :

`int` : nombres entiers. On utilse souvent la notation `short`. Nous verrons plus loin la différence; sachez dès mair tenant qu'enTurbo C elle est identique à `int`.

`float` : nombres réels

`char` : caractères.

LES DECLARATIONS DE VARIABLES ET LE TYPE INT

Les variables et constantes entières en langage C ont un rôle prépondérant. Contrairement à beaucoup de langages, C travaille en effet par défaut en mode entiers. C'est-à-dire que s'il rencontre dans le programme un nombre sans précision de type, le compilateur le considère comme un entier. Ce phénomène survient également lors de l'exécution.

Un nombre entier en C se présente donc comme une suite de chiffres ou d'opérations travaillant sur des entiers. Voici quelques exemples de nombres entiers et non entiers en Turbo C :

```
12 +12 -465  32000 32767
```

Tous ces nombres sont des entiers du type int.

`32768`: ce nombre est un entier mais dépasse le format int

`32.0` : n'est pas un entier car il comporte un point décimal.

`1/3` : contrairement au Pascal, cette expression est du type entier car "1" et "3" sont du type entier. Elle a donc la valeur 0.

`1.0/3` : Cette expression est un réel et non un entier, le "1.0" forçant le type réel. L'expression a donc la valeur 0.333333. Toute expression comportant un réel donne un résultat réel.

`(int)1.0/3` : Ceci s'appelle un "CAST" en langage C. En utilisant un cast, on force le type de l'opérande qui le suit. Cette expression équivaut donc à 1/3, c'est par conséquent une expression du type entier. Le cast permet d'éviter les ambigüités au niveau des types d'expressions; nous les utiliserons assez souvent dans les expressions réelles.

`1.0/(int)3`: Malgré le cast, cette expression a un type réel. Avez vous compris pourquoi ? Si non, reportez vous à l'exemple précédent pour l'explication du cast.

Pour utiliser une variable du type entier, il faut la déclarer quelque part. Pour cela, nous avons deux possibilités.

En effet, le langage C autorise deux endroits pour la déclaration des variables. Le premier est "au-dessus des fonctions", ce qui signifie généralement au tout début du programme. Les variables déclarées à cet endroit sont dites GLOBALES car elles sont accessibles et modifiables par toutes les fonctions, y compris bien entendu la fonction `main`. Voici un exemple de déclaration de variable globale :

```
int  x,y;
main()
{

}
```

Dans cet exemple fictif, les variables x et y de type entier `int` sont disponibles dans toutes les fonctions du programme. Les variables globales sont exactement identiques à celles du Pascal. Toutefois la syntaxe de Turbo C est la suivante : le type, puis la liste des variables déclarées, séparées par des virgules, et terminée par un point-virgule.

Une variable est connue par son IDENTIFICATEUR, c'est-à-dire son nom. En C standard, les caractères admis sont les lettres (minuscules ou majuscules), les chiffres et le souligné, "_". La variable doit commencer par une lettre ou le souligné. Seuls les 6 premiers caractères sont pris en compte : les variables CARACTERE_1 et CARACTERE_2 sont donc identiques. Par contre, les majuscules et minuscules sont différenciées. Ainsi, les variables `Toto` et `toto` sont-elles différentes.

Turbo C apporte trois améliorations : tout d'abord, le symbole Dollar "$" est utilisable au même titre que le souligné (sauf en début de nom). D'autre part, les noms de variables comportent jusqu'à 32 caractères, tous significatifs. Enfin, Turbo C peut, sur demande, ignorer les différences entre majuscule et minuscule.

Toutefois nous ne recommandons pas cette dernière option. En effet elle isole vos programmes de tous les autres compilateurs C ne disposant pas de cette option. De même, l'utilisation d'identificateurs de 32 caractères n'est guère recommandée. Nous vous conseillons d'utiliser des identificateurs longs et explicites, certes, mais dont les 6 ou 7 premiers caractères diffèrent. Les deux exemples suivants sont aussi parlants l'un que l'autre :

```
prix_taxes_comprises
prix_taxes_omises

sans_taxes;
avec_taxes;
```

Le second exemple sera préféré car, en apportant la même clarté, il reste compatible avec les standards les plus répandus.

Turbo C offre également une possibilité supplémentaire, reconnue par tous les compilateurs C : l'initialisation automatique des variables. Nous pouvons par exemple donner à x la valeur 1 et à y une autre valeur de la façon suivante :

```
int  x=1,y=x+46;
main()
{...}
```

L'initialisation de y comporte une opération tenant compte de la valeur de x. Attention, cela est très pratique mais veillez à ce que les variables utilisées dans une initialisation soient elles-mêmes initialisées avant ! Par exemple, le programme suivant donne une valeur indéfinie à la variable y :

```
int  x,y=x+1;
main()
{....}
```

La variable y aura bel et bien la valeur x+1, mais personne ne pourrait prévoir ce que vaut x, ni donc y.

Les initialisations sont intéressantes pour simplifier la fonction main; en effet, elle n'a plus besoin alors de procéder à l'initialisation des variables globales comme c'est généralement le cas (par exemple en Turbo Pascal).

Le deuxième endroit où nous pouvons déclarer des variables est l'intérieur des fonctions. Dans ce cas, les variables sont dites LOCALES justement parce qu'elles sont localisées dans cette fonction et ne peuvent être utilisées que par celle-ci.

Par exemple, si nous désirons que x et y ne soit pas globales mais locales à la fonction main, voici ce que devient l'exemple :

```
main()
{
  int  x=1,  y=45;
  . . .
}
```

Dans ce cas précis, cela revient au même. Mais si le programme comportait une autre fonction, elle ne pourrait pas utiliser x et y. Par exemple, le programme suivant ne peut pas être compilé :

```
carre()
{
   y=x*x;
}
main()
{
  int  x=1,y;
  carre;
}
```

En effet, la fonction carre (vous remarquerez que sa structure est totalement identique à celle de la fonction main, puisque c'est la structure générale d'une fonction) ne connait pas les variables x et y. Cet exemple est donc incorrect.

C'est aussi bien, car c'est une hérésie d'un point de vue clarté : la lecture de la fonction main permet de constater que nous ignorons ce que fait la fonction carre : peut-être modifie t-elle `x`, ou bien `y` ? C'est un excellent exemple de ce qu'il ne faut pas faire.

Pour que cet exemple fonctionne, il faut placer `x` et `y` en variables globales, ce qui donne le programme suivant :

```
int x=1, y;

carre()
{ y=x*x }

main()
{ carre() }
```

Cet exemple fonctionne, mais il est encore pire car nous ignorons totalement ce que fait le programme principal `main()` ! Nous verrons plus tard comment utiliser les fonctions d'une façon plus propre et surtout plus pratique.

Tout ce que nous venons d'apprendre sur les déclarations de variables (globales, cast, locales, initialisations) est valable pour tous les types de données disponibles en C, y compris ceux créés par l'utilisateur comme nous le verrons un peu plus tard (on ne peut pas tout apprendre à la fois, n'est-ce pas ?)

LE TYPE FLOAT

Les nombres réels sont également appelés flottants car ils possèdent une virgule flottante. Cette appelation date des temps héroïques où les ordinateurs (ou les langages) ne pouvaient travailler qu'avec un nombre de décimales (chiffres après la virgule) défini. Par exemple, toutes les opérations se faisaient avec deux chiffres après la virgule. Depuis l'invention du codage "mantisse + exposant", le nombre de décimales n'est plus fixe, la virgule est donc "flottante" puisqu'elle peut changer de position. Par extension de langage, des virgules, ce sont les nombres qui sont devenus flottants ! Il s'agit d'une dénomination presqu'aussi ancienne que celle des bugs !

En langage C, un nombre flottant est une suite de chiffres comportant obligatoirement un point décimal (à la place de la virgule, car Turbo C ne renie pas ses origines outre-atlantiques). Il est aussi possible de les exprimer sous forme scientifique, soit une mantisse et un exposant. Voici quelques exemples de nombres `float` :

13.24

-3.14159

2.71828

0.5625

5625E-4 : ce nombre est le même que le précédent. Eh oui !

56.25E-2 : c'est encore le même !

.05625E1 : .... Devinez !!!

32E+46 : ce nombre est un flottant mais il dépasse le format `float` car il est trop grand (exposant). Il provoquera une erreur du type overflow (dépassement de capacité).

15E-76 ce nombre est un flottant, mais il dépasse le format `float` car il est trop près de zéro (exposant trop grand en valeur absolue). Même commentaire que le précédent.

1/3 cette expression a une valeur entière ! Voir le paragraphe sur les nombres de type `int` !

(float)1/3 Le cast est tout à fait utilisable ici aussi. Dans cet exemple, il force la division au format flottant. L'expression a donc la valeur 0.333333.

1/(float)3 Ici aussi le résultat est de type flottant. En effet, le compilateur C travaille par défaut en `int`, mais il suffit qu'un opérande ait un type non entier pour que l'opération qui l'utilise ait le même type. Dans notre cas, la division est de type flottant car le diviseur est forcé au format `float`.

Les variables de type float sont déclarées aux mêmes endroits que les variables int. Comme nous l'avons déjà souligné, les types de données, quels qu'ils soient, ont tous les mêmes droits et les mêmes obligations. Vous pouvez donc utiliser des variables flottantes globales ou locales, initialisées ou non. Voici un petit exemple purement illustratif :

```
float globale=1.0;
main()
{
  float locale;
  float initialisee=12.0;
  float autre=1/3.5+76, encore=initialise+autre*2.0-45;
  float danger=1/locale;
  float attention=1/3;
  float badaboum=1/attention;
  float correct=1.0/3;
  float vaut_3=1/correct;
  ....
}
```

Dans cet exemple, la variable `globale` est initialisée. La variable `locale` ne l'est pas, sa valeur sera donc indéterminée. Par contre, les variables `initialisee` et `autre` auront, depuis la ligne de leur déclaration, une valeur

de départ. La variable `encore` a elle aussi une valeur bien définie. Mais la variable `danger` porte bien son nom car si par hasard (et par malheur) la variable `locale` contient 0, alors l'initialisation de `danger` peut avoir un effet réellement indéterminé. Encore une fois, si vous utilisez une variable dans une initialisation, assurez-vous que cette variable soit elle-même initialisée auparavant. La variable `attention` est bien définie, mais comme vous l'avez peut-être deviné, elle contient la valeur 0 car la division de l'initialisation est au format entier. Conséquence : la variable `badaboum` provoque .. un badaboum ! L'erreur de division par zéro ne sera détectée que lors de l'exécution. Rassurez-vous, cela ne bloquera pas votre ordinateur car aujourd'hui les micro-processeurs eux-mêmes détectent les divisions par zéro, mais votre programme sera stoppé.

La même initialisation, correcte cette fois, est illustrée par les déclarations et initialisations de `correct` et `vaut_3`. `Correct` aura la valeur flottante 0.33333, et `vaut_3` contiendra par conséquent la valeur 3.0, car `1/correct` est une expression du type flottant (la variable correct force le type de l'expression au format `float`).

Enfin, une dernière remarque s'impose : Turbo C comporte un type double précision, `double`. Par définition du langage C, les expressions de type flottant sont, sauf dans les cas précisés, toutes calculées en double précision. Elles sont ensuite, le cas échéant, ramenées au format voulu (entier, float...). Nous n'utiliserons pas le type double pour l'instant, sachez seulement que cette nuance ne change absolument rien aux limites du type `float` et aux règles concernant les expressions flottantes. Par contre, elle ralentit les calculs sur les nombres réels, mais on ne peut pas tout avoir, précision et vitesse à la fois !

LE TYPE CHAR

C possède un type caractère assez particulier. En effet, il distingue deux types de caractères : les caractères ASCII, et les caractères dits non imprimables. Dans tous les cas, sauf un, les caractères ASCII sont exprimés directement entre apostrophes : 'A' '&' '7' '(' 'r' sont des caractères ASCII.

Le cas particulier que nous avons cité est celui du caractère "backslash", à savoir '\'. Ce caractère sert de PREFIXE aux caractères non imprimables. Un caractère nonimprimable est généralement quelque chose que vous ne pouvez pas taper au clavier, comme par exemple le passage à la ligne (touche ENTER). Dans ce cas, le langage C vous permet tout de même de stocker ces caractères grâce au préfixe '\', suivi soit d'un code "escape", soit d'un code numérique. Les codes "escape" sont des codes mnémoniques associés aux caractères les plus utilisés. Voici les caractères non imprimables les plus utiles et leur équivalent en Turbo Pascal, pour ceux qui connaissent ce dernier.

`'\n'` représente le passage à la ligne. En Pascal : #$A#$D ou ^M^A.

`'\b'` représente le retour en arrière. En Pascal : #8 ou ^H.

`'\a'` représente le BIP (Controle-G). En Pascal : #7 ou ^G.

`'\\'` permet d'utiliser le Backslash lui même.

`'\''` représente l'apostrophe . En pascal : ""

`'\"'` représente les guillemets

`'\t'` représente la tabulation. En Pascal : #9 ou ^I

Les caractères non imprimables peuvent aussi être représentés par un code numérique hexadécimal de la façon suivante :

`'\xnn'`, où "x" est le X minuscule et "nn" le code hexadécimal du caractère.

Par exemple, les caractères suivants sont identiques :

`'\x07'` et `'\a'` : le caractère qui fait BIP !

`'\x0A'` et `'\n'` : le passage à la ligne.

`'\x41'` et `'A'` : c'est bien un A, malgré son allure inquiétante !

L'utilisation de ces caractères non-imprimables ne doit pas vous effrayer. Il s'agit uniquement d'un moyen pratique de représenter des caractères qui ne sont normalement pas utilisables au sein d'un texte ! Le plus utile est `'\n'` qui permet de passer à la ligne. Nous l'avons déjà rencontré dans HELLO.C.

Les variables de type `char` répondent aux mêmes lois que celles des autres types. On peut donc déclarer une variable locale ou globale, initialisée ou non :

```
char globale='\n';
main()
{
  char locale;
  char initialisee='C';
  char erreur='\xFF'+initialisee;
}
```

Toutes ces variables sont correctes sauf celle appelée `erreur`. En effet, une variable caractère ne peut contenir qu'un seul caractère. Mais le compilateur ne vous renverra pas d'erreur. Pourquoi ?

En C, les caractères sont en réalité compatibles avec les entiers. En clair, ils peuvent être utilisés soit pour le caractère qu'ils représentent, soit pour son code. Dans notre initialisation, la variable erreur va donc recevoir non pas deux caractères accolés, mais un seul caractère qui sera celui dont le code est la somme des deux autres. En termes d'entiers, le nombre hexadécimal FF vaut -1. La somme de -1 et du code du caractère 'C' nous donnera donc le code du caractère 'B', qui précède C dans le code ASCII. Notre variable erreur contiendra donc le caractère 'B'.

Cette compatibilité apparemment dangereuse est en fait à la base d'une très grande souplesse dans la manipulation des caractères, et surtout d'une très grande rapidité. Nous aurons l'occasion d'y revenir en détail.

LES TABLEAUX

Comme ses confrères, le langage C autorise bien entendu la réunion sous un même nom de plusieurs variables de même type. On appelle un tel ensemble TABLEAU. Un tableau se déclare comme une variable à part entière, mais une indication supplémentaire permet de connaitre sa taille (le nombre de variables qu'il contient), et son type (qui est le type de ces variables).

Ainsi, la déclaration suivante :

```
int   tablo [20];
```

permet de déclarer un tableau de 20 entiers. Pour utiliser une des cases d'un tableau, il suffit d'indiquer son indice entre crochets :

```
tablo [4]=12;
```

affecte la valeur 12 à la case 4 du tableau.

Ceci amène une remarque très importante. : les tableaux commencent toujours à l'indice 0 (zéro). Ce n'est pas gènant en soi, mais vous devez vous souvenir que dans la déclaration, la taille est véritablemant la TAILLE du tableau, et non la plus grande valeur que peut prendre l'indice. Cela signifie pour notre exemple que nous pouvons utiliser les cases `tablo [0]` jusqu'à `tablo [19]`, et que `tablo [20]` n'existe pas.

Vous devrez faire attention à cela lors de vos débuts car l'erreur est vite faite, surtout pour ceux qui ont l'habitude du BASIC ou du Pascal.

Un tableau peut être de type tout à fait quelconque, y compris de type tableau. La déclaration d'un tableau est en réalité constituée par les opérateurs crochets. Vous avez pu remarquer en effet, si vous programmez en Pascal, l'absence d'un mot clé comme `array`. Voici des déclarations identiques en Pascal et en C (on remarque en passant la concision du langage C, moins "verbeux" que Pascal). Notez la différence des constantes définissant les tailles:

PASCAL:

```
- - - -
VAR     tablo : ARRAY [0..19] OF INTEGER;
        i: INTEGER;
```

Turbo C:

```
- - - -
int tablo [20];
int i;
```

Puisqu'il suffit d'ajouter des crochets et une taille pour déclarer un tableau, nous pouvons également faire tout aussi simplement des tableaux à plusieurs indices de la façon suivante :

```
int
tablo [10] [5];
float reels [359] [2];
```

Une déclaration équivalente en Pascal pour le tableau d'entiers sera par exemple :

```
VAR tablo : ARRAY [0..9,0..4] OF INTEGER;
```

ou

```
VAR tablo : ARRAY [0..9] OF ARRAY [0..4] OF INTEGER;
```

Du point de vue définition, le tableau `tablo` de notre exemple en C est un tableau [10] d'un tableau [5] d'entiers et non l'inverse. Pour utiliser une des cases, nous utiliserons toujours la même notation, logique et simple :

```
tablo [indice entre 0 et 10] [indice entre 0 et 5]
```

Le nombre d'indices d'un tableau n'est pas limité. La déclaration suivante est correcte :

```
int octet [2] [2] [2] [2] [2] [2] [2] [2];
```

Ce tableau peut contenir 256 entiers, les indices de chaque dimension pouvant prendre les valeurs 0 et 1.

Notez que les tableaux en langage C sont tout de même moins souples, au niveau des indices autorisés, que ceux du langage Pascal : ainsi, en C, les tableaux ont-ils toujours des indices entiers positifs et commençant à la valeur 0. En Pascal, nous pouvons par exemple utiliser les tableaux suivants :

```
VAR     Joli : ARRAY ['A'..'Z'] OF REAL;
        Beau : ARRAY [-15..0] OF BOOLEAN;
```

Ce genre d'indice est interdit en C, mais vous apprendrez vite à vous en passer grâce à la plus grande souplesse de C quant aux types de tableaux et de données.

Enfin, Turbo C autorise également l'initialisation des tableaux lors de leur déclaration. Les valeurs doivent alors être en nombre identique à la taille du tableau (y compris la case 0 bien entendu) et du type correct. On les place entre accolades et séparées par des virgules. S'il y a plusieurs tableaux, toujours avec le même principe, il s'agira d'une liste de valeurs composées elles-mêmes de listes de valeurs ! Voici trois exemples :

```
int  carres [5]  = {0,1,4,9,16};
int  tableau [3] [2] = { {00,01}, {10,11}, {20,21} };
int  tableau [3] [2] = { 0,1,10,11,20,21 }
```

Notez que les deux derniers exemples sont tout à fait équivalents, on choisira le style de notation approprié. Nous conseillons d'utiliser le premier, car il a l'avantage de mettre en évidence la structure du tableau. Mais c'est une question de goût, comme beaucoup de choses en C.

Les tableaux de type char, ou "chaînes de caractères"

Comme tous les langages modernes, le langage C propose sa version des chaînes de caractères. Sous ce terme se cache en fait un type de donnée qui peut recevoir des phrases, ou des expressions comportant plusieurs caractères accolés.

En C, le type string du Pascal (type qui ne fait d'ailleurs pas partie des définitions standard du Pascal, mais qui a fini par s'imposer de lui-même) n'existe pas. Théoriquement, la gestion des phrases / chaînes de caractères passe par des tableaux de caractères. Nous allons brièvement nous y attacher, mais le sujet est tellement vaste que nous y reviendrons plus en détail.

Vous devez tout de même retenir que le langage C a évolué par rapport à sa naissance, et qu'une certaine forme de string a fini par apparaître. En fait, une extension de la "librairie standard" apporte les fonctions nécessaires à la gestion de chaînes de caractères, ce qui permet de ne pas se soucier de ce problème. Les fonctions en question sont maintenant admises comme relativement standard, et peuvent être utilisées sans crainte de perdre la portabilité.

Un tableau de caractères se déclare de la même façon que les autres tableaux, avec une taille entre crochet derrière un nom de variable.

```
char  phrase [80];
```

Cet exemple déclare un tableau de 80 caractères.

Le type chaîne de caractères est un type à part entière, et il possède sa propre notation. Alors que les caractères, nous l'avons vu, sont placés entre apostrophes, les chaînes sont placées entre guillemets. Vous devez tout prix, dès maintenant, faire la distinction qui est très importante :

`"Gaston"` est une chaîne de caractères.

`'G'` est un caractère

mais également :

`'A'` est un caractère.

`"A"` est une chaîne de un caractère.

Cette nuance subtile est vitale car si vous tentez par exemple de stocker un caractère dans une variable de type chaîne, vous provoquerez une erreur :

```
char   phrase [20]='c'; <-- ici, une grosse erreur !

char   lettre = "c" ; <-- ici, la même erreur dans l'autre sens !
```

Mais rassurez-vous, il est possible de faire des opérations associant les deux types. Par exemple :

```
phrase [indice] = lettre;
```

Nous reviendrons plus en détail sur les manipulations des chaînes de caractères.

RESUMONS-NOUS

Nous venons de voir trois types de données de base :

- `int` qui représente les nombres entiers.
- `float` qui représente les nombres réels (flottants).
- `char` qui représente les caractères.

Et nous avons découvert la structure tableau:

- `int[]` pour les tableaux d'entiers.
- `float[]` pour les tableaux de réels.
- `char[]` pour les tableaux de caractères.

Nous avons de plus appris que ce dernier type de tableau constitue aussi un type de données à part entière, appelé chaîne de caractères (ou simplement chaîne).

Enfin, nous avons appris où et comment déclarer des variables de ces types, en les initialisant au besoin, et quelle est la forme des constantes.

Si certains points ne vous paraissent pas tout à fait clairs, n'hésitez pas à parcourir de nouveau ce chapitre avant de passer à la suite.

CHAPITRE 4

PRINTF ET SCANF

Avant d'attaquer la programmation proprement dite, il nous reste deux éléments importants à examiner : la fonction qui permet d'afficher des résultats sur un écran et celle qui permettra de saisir des données au clavier.

LA FONCTION PRINTF

Nous avons entrevu la fonction `printf` à plusieurs reprises. Vous savez déja qu'elle permet d'afficher un texte directement comme le PRINT du BASIC et le WRITE du Pascal. Elle peut faire beaucoup plus, bien entendu.

La syntaxe classique de la fonction `printf` est la suivante, sous une forme très simplifiée :

```
printf ("format",variables);
```

Par format, nous entendons une chaîne de caractères comportant certains symboles spéciaux. En effet, `printf` a besoin, avant d'afficher des variables ou des données, de connaître leur type et leur taille à l'écran. Pour cela, il faut préciser ces paramètres dans le format.

Un exemple simple :

```
printf ("Nombre = %d , Carre = %d \n", 23, 23*23);
```

Examinons le format. La chaîne comporte un `\n`, que nous connaissons déjà. Nous savons qu'il provoque le passage à la ligne. Le texte normal comme `Nombre =` et les blancs ne posent pas de problème non plus. En revanche, nous devons nous pencher d'avantage sur les `%d`. Contrairement à ce que l'on pourrait croire, nous n'obtiendrons pas ceci à l'écran :

```
Nombre = %d , Carre = %d
```

mais l'affichage suivant :

```
Nombre = 23 , Carre = 529
```

En effet, dans la chaîne de caractères placée dans le `printf`, les `%d` indiquent tout simplement à la fonction qu'à cet endroit prendra place un nombre au format entier `int`. Nous avons deux emplacements dans la chaîne, et deux expressions entières ensuite. Elles seront calculées, et le résultat figurera dans les emplacements correspondants, dans l'ordre d'apparition.

D'autres symboles sont nécessaires pour afficher les autres types de donnée. Voici ceux qui nous concernent pour l'instant :

`%d` emplacement pour un nombre de type int.

`%f` emplacement pour un nombre de type float.

`%c` emplacement pour un caractère (type char).

`%s` emplacement pour une chaîne de caractères.

`%%` affiche un "%" (il faut tout de même pouvoir l'obtenir !)

Notez d'autre part que vous pouvez également formater le nombre ou la donnée à l'intérieur de cet emplacement. Voici quelques exemples :

`%4d` : ce nombre entier sera cadré dans 4 colonnes sur l'écran.

`%5.2f` : ce nombre flottant sera affiché avec 5 chiffres avant la virgule et 2 après.

`%20s` : cette chaîne de caractères sera tronquée ou complétée à 20 caractères.

`%9c` : ce caractère sera cadré à droite dans 9 colonnes (les 8 premières seront remplies de blancs).

Si la donnée ne correspond pas au format indiqué dans l'emplacement, la fonction fera son possible pour l'afficher tout de même. Ainsi, un nombre entier placé dans un emplacement de flottant sera-t-il transformé en flottant avant l'affichage, et inversement. Par exemple,

```
printf("%d",1.0/3)
```

affichera "0".

Pour afficher des variables, le principe est aussi simple. Tapez et exécutez le programme suivant :

```
main()
{
  int   i,j;i=1;
  j=i+1;
  printf("i=%d\nj=%d\nj+1=\n",i,j,j+1);
}
```

Certes, la chaîne "format" n'est pas très lisible. Mais vous obtiendrez sur l'écran la sortie suivante :

```
i = 1

j = 2

j+1=3
```

Malgré l'apparence parfois barbare des chaînes de format pour `printf`, rassurez-vous, on s'y fait très vite. Notez que l'équivalent en Pascal ne serait pas tellement plus lisible :

```
WRITELN('i=',i,^M^A'j=',j,^M^A'j+1=',j+1);
```

C'est peut-être même pire !

Il nous reste à examiner le cas de `scanf`, qui est au clavier l'inverse de ce que `printf` est à l'écran. `printf` était une fonction de sortie, `scanf` est une fonction d'entrée.

LA FONCTION SCANF

`scanf` agit donc de manière symétrique à `printf`. Elle lit depuis la clavier la valeur à donner à une ou plusieurs variables. Sa syntaxe, également simplifiée, est la suivante :

```
scanf("symboles", adresses);
```

Les symboles sont les mêmes que ceux utilisés par `printf` : `%d`, `%f`, `%c` pour représenter un entier, un réel, un caractère, etc. Par contre `scanf` s'attend à ce qu'on lui indique, non pas le nom de la variable à lire, mais l'adresse de celle-ci. Cette distinction est essentielle. Pour bien la comprendre, nous allons faire une petite digression.

Un peu d'adresses !

Tout ordinateur possède de la mémoire. Elle est en général répartie en deux zônes; l'une, la mémoire morte ou ROM, ne peut être que lue (on ne peut pas y écrire). Elle contient les informations de base nécessaires au fonctionnement de la machine. C'est elle qui se charge par exemple du démarrage lors de la mise sous tension. Comme on ne peut pas la modifier, Turbo C ne peut rien y stocker et elle ne nous intéresse pas beaucoup ici.

L'autre partie est la mémoire vive ou RAM. C'est là que sont chargés les programmes et les données (depuis le disque qui les conserve entre deux sessions).

L'unité d'information est le bit. On peut le comparer à un interrupteur microscopique qui est soit ouvert, soit fermé. Lorsqu'il est fermé, le courant y passe, tandis que lorsqu'il est ouvert, le courant est stoppé. Ces deux états peuvent donc représenter les deux chiffres 0 et 1 (d'où le nom bit, contraction de Binary Digit, chiffre binaire).

A l'intérieur de la mémoire, les bits sont regroupés par paquets de huit : huit bits forment un octet. Les combinaisons de ces huit bits donnent 256 valeurs posibles à un octet.

Nous conseillons au lecteur débutant de se familiariser avec l'arithmétique binaire (et l'arithmétique hexadécimale qui lui est souvent associée) à travers un des très nombreux ouvrages spécialisés traitant de cette question.

Le contenu d'un octet est toujours un nombre (compris entre 0 et 256); mais la façon dont on interprète ce nombre peut varier considérablement. Ce peut être une valeur numérique, un caractère (codé par exemple suivant le code ASCII), un code machine exécutable, ou tout autre type de donnée.

Chaque octet occupe une place unique au sein de la mémoire, repérée par son adresse, c'est à dire son numéro d'ordre. On peut comparer l'adresse d'un octet au numéro d'un étage, ou d'une maison dans une rue. On dira par exemple que l'octet d'adresse X content la valeur Y.

Suivant le type de donnée à mémoriser, les octets sont souvent regroupés eux-mêmes en mots (deux octets), en longs mots (4 octets), en paragraphes (16 octets) ou en blocs de longueur déterminée. Chacun de ces groupes aura lui aussi une adresse, en fait celle du premier octet qui le constitue.

Lorsque vous faites une déclaration comme

```
int i = 25;
```

le compilateur sait que la variable i occupera deux octets (car un entier est codé sur deux octets). Il réserve donc une place en mémoire, longue de deux octets, dans laquelle il range la valeur assignée à i (25), et se souvient de l'adresse de rangement. Lorsque plus tard dans le programme une nouvelle assignation sera faite, par exemple `i = 103`, il changera la valeur contenue dans les deux octets représentant la variable `i`.

Aux yeux du programme, i représente bien le contenu de la variable `i`; l'adresse de cette variable, c'est à dire celle du premier des deux octets qui la contiennent, est à priori inconnue du programmeur. Elle est déterminée par le compilateur, suivant l'emplacement de la déclaration, le modèle mémoire en vigueur, et la place disponible.

Cependant, il arrive parfois, et c'est même fréquent en C, que l'on ait besoin de connaître cette adresse. C'est précisément le cas avec `scanf`. Il existe donc un opérateur spécial qui renvoie l'adresse d'une variable. Cet opérateur se note "`&`": si on a déclaré une variable i comme ci-dessus,

```
&i
```

désigne l'adresse de cette variable.

Nous reviendrons en détail sur ces notions un peu plus loin. Pour l'instant, nos pouvons écrire notre fonction `scanf` :

```
scanf("%d",&i);
```

Cet exemple lira au clavier une valeur entière (puisque nous avons indiqué `%d`), et rangera celle-ci à l'adresse de la variable `i`. Cette dernière prend donc la nouvelle valeur. Bien entendu, la variable `i` doit avoir préalablement été déclarée de type entier, sinon gare aux erreurs!

Et voici notre premier programme interactif :

```
main()
{
  int  i,j;
  printf("Entrez un nombre entier ");
  scanf("%d",&i)
  j=i*i;
  prinf("\nLe carré de %d est %d.",i,j);
}
```

Avec `printf`, on peut imprimer plusieurs variables dans une même chaîne de caractères; avec `scanf`, on peut aussi lire plusieurs variables d'un seul coup. En ce sens, la chaîne entre guillemets sert aussi de formatage. Par exemple,

```
scanf("%d %d,&a,&b);
```

où les deux symboles `%d` sont séparés par un blanc, acceptera deux entiers, séparés au clavier par un nombre quelconque d'espaces, de tabulations ou de retour-chariots. Mais si l'on écrit :

```
scanf("%d,%f",&a,&r);
```

on lira au clavier un réel suivi d'un entier, séparés cette fois par une virgule.

Lentement, mais sûrement, nous avançons! Nous savons dès maintenant lire et afficher des variables, ce qui est la base de tout programme. En fait, ce n'est pas si compliqué. Quoique...il y a quand même un cas particulier, celui des chaînes caractères. Avant de pouvoir les aborder, il faut, comme promis, revenir sur les notions d'adresses.

C'est ce que nous allons faire dans le prochain chapitre.

CHAPITRE 5

LES POINTEURS

Le langage C est très friand d'adresses, et nous aurons souvent besoin de les manipuler. Nous avons vu un peu plus haut qu'il existe un opérateur qui renvoie l'adresse d'une variable. Souvenez-vous, si `truc` est une variable préalablement déclarée, `&truc` désigne son adresse (déterminée par le compilateur).

LE TYPE POINTEUR

Mais dans quoi allons nous stocker cette valeur? Une adresse n'est ni un entier, ni un réel, ni un caractère! Et bien dans un type spécial de variable, créé tout exprès pour contenir des adresses. Ce type se nomme pointeur. Nous allons voir dans un moment comment déclarer une telle variable; supposons pour l'instant que ptr est déclarée de type pointeur; on pourra donc écrire :

```
ptr = &i;
```

`ptr` contient l'adresse de `i`; en quelque sorte, il "montre `i` du doigt"; on dit qu'il POINTE sur `i`. `ptr` est un moyen indirect d'atteindre `i`; c'est pourquoi on dit qu'un pointeur est une indirection.

Cette notion est essentielle en C, comme d'ailleurs dans la plupart des langages. Le premier exemple (que nous avons déjà entre-aperçu) concerne les tableaux et par voie de conséquence les chaines de caractères, qui n'en sont qu'un cas particulier :le nom du tableau contient l'adresse de celui-ci; c'est donc un pointeur sur le tableau. Nous verrons qu'en C, tableaux et pointeurs sont souvent équivalents.

Une autre remarque s'impose aussi tout de suite : partout où une adresse est admise, un pointeur le sera aussi, puisqu'il contient une adresse.

A quoi cela sert-il? pourquoi manipuler des adresses, alors qu'on a les données elles mêmes sous la main? Pour différentes raisons. En changeant l'adresse contenue dans un pointeur, on atteindra immédiatement n'importe quelle donnée. L'usage des pointeurs permettra par exemple de construire des structures d'arbre ou de liste chaînée, dans lesquelles chaque élément contient un pointeur sur le suivant. On pourra ainsi parcourir la liste, la trier ou en extraire des éléments simplement par la modification de quelques pointeurs, sans déplacer effectivement les données volumineuses : gain de place, gain de temps.

Une autre bonne raison est que certaines fonctions standard de Turbo C s'attendent à trouver en paramètre l'adresse de leurs opérandes; on leur passera donc des pointeurs sur ces opérandes. C'est par exemple le cas de `scanf`, comme nous l'avons vu plus haut.

Mais ce n'est pas tout. Tout comme en Pascal, les pointeurs permettent la création dynamique de variables par le programme lui-même : on demande l'allocation de la place mémoire requise par la nouvelle variable, et on en stocke l'adresse dans un pointeur. C'est ainsi que l'on procèdera lorsqu'on ne connaît pas au moment de l'écriture du programme le nombre de variables qu'il utilisera (une gestion de fiches par exemple).

Nous verrons aussi d'autres emplois des pointeurs, par exemple pour accéder à une partie interne d'une donnée structurée.

La déclaration des pointeurs

Chose promise, chose due! voici comment déclarer une variable pointeur : on fait précéder le nom de la variable pointeur du signe *.

```
int  *ptr;
```

Ceci revient à dire "la variable `ptr` contiendra l'adresse d'un entier". En Pascal on aurait écrit :

```
ptr : ^integer;
```

Nous pouvons illustrer ceci d'un tout petit exemple:

```
main()
{
   int nb, *nbptr;
   nb = 20;
   nbptr = &nb;
  printf("Le nombre %d est stocké à l'adresse %p",nb, nbptr);
}
```

La première ligne déclare à la fois une variable entière `nb` et une variable pointeur `nbptr`, qui pointera sur un entier. Après avoir affecté une valeur à `nb`, on fait pointer `nbptr` sur `nb` (`nbptr = &nb`) en assignant à `nbptr` la valeur de l'adresse de `nb`.

Il ne reste plus qu'à afficher les deux valeurs : c'est le rôle du `printf`, que vous commencez à bien connaître. Juste une remarque à ce sujet : l'entier `nb` est affiché par le symbole `%d`; le pointeur, lui, demande un symbole nouveau, car c'est un nouveau type de donnée. Le symbole d'affichage d'un pointeur est `%p` (facile à mémoriser!).

RESUMONS-NOUS.

Lorsqu'on a une variable `v`, l'adresse de cette variable est donnée par l'expression `&v`, que l'on stocke dans une variable pointeur ptr (on dit simplement un pointeur), déclarée par `*ptr`.

L'INDIRECTION

Reprenons le petit programme ci-dessus, et considérons la ligne

```
nbptr = &nb;
```

`nbptr` pointe à présent sur `nb` : il en contient l'adresse. Si nous pouvions modifier le contenu de l'adresse pointée par `nbptr`, nous agirions directement sur `nb`. Il existe bien sûr un opérateur aussi pour cela : c'est l'opérateur d'indirection, qui se note *, et que nous venons de voir pour la déclaration de pointeur :

```
*nbptr
```

représente le contenu de l'adresse pointée par nbptr, c'est à dire nb lui-même. Bien sûr, ici cela n'a pas grand intérêt, puisque nous avons accès à nb lui-même. Mais il y aura fréquemment des cas où nous ne connaitrons que l'adresse (le pointeur) d'une donnée, sans que celle-ci soit directement accessible. L'opérateur d'indirection devent alors très précieux, car il permet d'agir sur la donnée pointée. Reprenons une fois encore le petit programme de la page précédente :

```
main()
{
    int nb, *nbptr;
    nb = 20;
    nbptr = &nb;
    printf("Le nombre %d est stocké à l'adresse %p",nb, nbptr);
    *nbptr = 50;
    printf("Maintenant, la variable nb vaut %d ",nb)
}
```

Nous avons ici modifié directement `nb` à travers son pointeur `nbptr`, par indirection.

LA CONVERSION DE TYPE

Revenons un instant sur la déclaration du pointeur. Nous avons écrit :

```
int *nbptr;
```

ce qui signifie que nbptr contiendra l'adresse d'un entier. Pourtant, une adresse est une adresse, et rien ne distingue celle d'un entier de celle d'un caractère ou d'un réel. Pourquoi faire ainsi la distinction? Parce que si vous

utilisez le pointeur pour modifier la valeur pointée comme nous venons de le faire, il est capital de connaître le type de donnée pointée.

Dans le cas présent, `nbptr` pointe sur un entier, qui occupe deux octets. Si, à l'aide de l'indirection, vous assignez une nouvelle valeur à `*nbptr`, celle-ci doit prendre la même place que l'ancienne, sous peine de déborder sur la donnée suivante en mémoire. La plupart des compilateurs vous interdiraient d'ailleurs d'écrire maintenant :

```
*nbrptr = "toto";
```

`toto` est une chaine qui occupe 5 octets (n'oubliez pas le caractère nul en fin de chaine), alors que `nbptr` est déclaré pointeur sur un entier (deux octets).

Toutefois... Turbo C, lui, acceptera cette instruction, mais vous préviendra par un message du type "assignation de pointeur non portable" (Non portable pointer assignation). Pourquoi l'autorise-t-il? parce qu'il y a des cas où un programmeur malin peut tirer profit de ce genre de confusion. Mais ceci est une autre histoire...

Il existe d'ailleurs un moyen "légal" de transformer un pointeur sur un type en un pointeur sur un autre type. On parle alors en bon français (!) de cast (ou casting), c'est à dire de conversion de type entre pointeurs. Il suffit pour cela de placer le type voulu, entre parenthèses, et suivi d'un espace et du signe `*`, devant le pointeur à transformer. Par exemple pour changer en pointeur sur un char en un pointeur sur un entier :

```
iptr = (int *) carptr;
```

Nous en verrons une application un peu plus loin.

CHAPITRE 6

LES CHAINES DE CARACTERES

DES CHAINES ONT DU CARACTERE!

Les chaînes, nous l'avons vu, sont définies en C comme des tableaux, ce qui est somme toute assez logique. Mais cela a plusieurs conséquences. Si l'on écrit :

```
char  phrase[20] = "Boujour à tous";
```

que va-t-il se passer? Le compilateur réserve quelque part en mémoire une place de 20 octets; il y place ensuite les caractères indiqués, et leur ajoute un octet nul (ASCII 0). Dans notre exemple, les 20 octets ne sont donc pas tous utilisés; mais cela signifie que `phrase[20]` pourra contenir en fait 19 caractères, qui auront les indices 0 à 18 dans le tableau, plus le caractère nul (indice 19). Souvenez-vous que les tableaux en C commencent toujours à l'indice 0.

Bon. Mais quel va être le contenu de la variable `phrase`? A première vue, on pourrait penser qu'elle contient réellement les mots "Bonjour à tous". En fait, il n'en est rien : souvenez-vous que le nom d'un tableau contient l'adresse du tableau. Le compilateur va donc placer dans `phrase` l'adresse du tableau, c'est à dire l'adresse de la première lettre, ici B. Par contre `phrase[0]` contient vraiment le caractère 'B', `phrase[1]` contient 'o', etc. En d'autres termes, la variable `tableau` contient l'adresse du tableau, tandis que les éléments indicés contiennent réellement les valeurs qui remplissent le tableau.

Cette particularité est très importante, et elle implique par exemple qu'on pourra lire directement une chaîne avec `scanf`, sans utiliser l'opérateur `&`, et écrire :

```
main  ()
{
    char  nom[20];
    printf("Quel est votre prénom?\n");
    scanf(%s,nom);
    printf("%s,  vous commencez à comprendre le C!\n",nom);
}
```

Notez le symbole `%s` pour représenter une chaîne, tant dans `printf` que dans `scanf`.

Une autre conséquence est l'interdiction de faire une assignation directe du type :

```
phrase = "J'aime le C!";
```

après la déclaration

```
char  phrase[20] = "Boujour à tous";
```

En effet, lors de la compilation d'une telle déclaration, la chaîne est stockée dans le code du programme, et `phrase` prend pour valeur l'adresse de cette chaîne. Les noms de tableaux sont des constantes qui ne peuvent être modifiées. L'instruction `phrase = "J'aime le C!";` signifierait en effet d'assigner à `phrase` l'adresse de la nouvelle chaîne `"J'aime le C!"` (créée quelque part dans le code). Elle est donc incorrecte, car cela reviendrait à modifier une constante. Le compilateur ne l'accepte d'ailleurs pas, et renvoie une erreur. Il faut donc trouver une autre façon de faire.

Nous allons nous servir de la fonction standard `strcpy`. Que fait cette fonction? Elle copie une chaîne donnée à l'adresse que vous spécifiez. Sa syntaxe est la suivante :

```
strcpy(adresse, chaine);
```

Pour modifier notre chaîne déclarée ci-dessus, nous écrirons donc :

```
strcpy(phrase,"J'aime le C!");
```

Que se passe-t-il ici? La chaîne `"J'aime le C!"` est créée quelque part dans le code, terminée par un caractère nul. `strcpy` copie un à un ses caractères, et les place à partir de `phrase` , c'est à dire à partir de l'adresse de notre variable chaîne. Les caractères sont copiés un par un, jusqu'à la rencontre du caractère nul. En sortie, le tableau `phrase[20]` contient bien la nouvelle chaîne. Mais, et c'est là le point délicat à bien comprendre, `phrase` n'est pas modifié : le tableau n'a pas changé d'adresse.

Pour reprendre le propos du chapitre précédent, le nom d'une chaîne est un pointeur sur le début de cette chaîne.

POINTEUR OU TABLEAU?

Dans ces conditions, il doit être possible de déclarer une variable chaîne comme un pointeur. C'est en effet une alternative : une chaîne est déclarée soit comme un tableau, soit comme un pointeur.

Considérez les deux déclarations suivantes :

```
char phrase[20];

char *mot;
```

Quelle différence y a-t-il entre les deux? La première déclare une chaîne sous forme de tableau, tandis que la seconde utilise un pointeur. Lorsqu'on déclare un tableau, la place de ce tableau est effectivement réservée en mémoire; on a donc le "droit" de remplir directement ce tableau; c'est ce que fait la fonction `scrcpy` que nous venons de voir.

Remarquez que dans le second cas, on ne spécifie pas la longueur de la chaîne. Pourquoi? Lorsqu'on déclare un pointeur, aucune mémoire n'est allouée à la nouvelle variable; il n'y a donc pas lieu d'indiquer sa longueur. Dans notre exemple, le compilateur sait simplement que `mot` pointera sur un caractère. Le pointeur contient donc au départ une adresse aléatoire.

Le piège à éviter est d'écrire, après la déclaration :

```
scanf("%s",mot);
```

Cette ligne est syntaxiquement correcte, mais est en fait très dangereuse, car `scanf` placera les caractères que vous tapez à l'adresse aléatoire contenue dans `mot`, sans se préoccuper de ce qui pouvait s'y trouver auparavant.Vous risquez donc d'écraser une autre donnée, ou pire une portion de programme, d'où plantage possible... et même à retardement : il n'aura lieu que lorsque la portion "polluée" sera exécutée, ce qui peut arriver longtemps après la saisie.

Quelle est la solution? Il faut initialiser le pointeur, en réservant la place mémoire nécessaire à la variable sur laquelle il pointe à l'aide d'une fonction spécialisée dans l'allocation mémoire, telle que `malloc` par exemple. En voici un exemple :

```
main()
{
    char *machaine;
    printf("Tapez votre prénom\n");
    machaine = (char*) malloc(20);
    scanf("%s",machaine);
    printf("Quelle chance, %s est justement");
    printf("mon prénom préféré!\n",machaine);
}
```

La seule ligne nouvelle concerne l'initialisation du pointeur:

```
machaine = (char*) malloc(20);
```

La fonction `malloc` réserve une zone en mémoire du nombre d'octets demandé, et renvoie un pointeur sur cette zone. Ce pointeur n'a pas de type particulier, car `malloc` ne sait pas à priori quel type de donnée va être rangée dans la zone qu'elle réserve. Il faut donc convertir ce pointeur indéfini en un pointeur sur un `char`; c'est le rôle de `(char*)`. Cette opération (la conversion de type), s'appelle casting, nous en avons déjà parlé au chapitre précédent.

Vous avez suivi? Continuons! Dans l'exemple ci-dessus, `machaine` est un pointeur sur le prénom tapé. Si on lui applique l'opérateur d'indirection, qu'allons nous obtenir? la première lettre du prénom. C'est logique : `machaine`

est un pointeur sur un caractère, donc `*machaine` est bien le caractère qui se trouve à cette adresse. Mais nous avons vu que ce premier caractère pouvait aussi être atteint par `machaine[0]`. Nous retrouvons ici la relation très intime qui existe en C entre tableaux et pointeurs. Si nous avons un tableau quelconque nommé par exemple `table`, nous aurons l'identité suivante :

```
*table = table[0]
```

Ces identificateurs se réfèrent tous deux au premier élément du tableau.

Mais ce n'est pas tout! nous pouvons aussi accéder aux autres éléments du tableau; de la même manière, nous aurons

```
*(table+1) = table[1]
*(table+2) = table[2], etc...
```

Pourtant, direz-vous, si `table` est l'adresse du premier élément, `table+1` doit être l'adresse suivante (un octet plus loin); que se passe-t-il si chaque élément du tableau a plus d'un octet de long (par exemple un tableau d'entiers, qui prennent chacun deux octets)? On va mesurer ici "l'intelligence" du compilateur C : par la déclaration du tableau, il connaît la longueur de ses éléments; lorsqu'on ajoute 1 à un pointeur, il ajoute en fait la longueur d'un élément. C'est ainsi que `table+1` pointe sur le second élément du tableau, et que `*(table+1)` est bien cet élément lui-même.

Plus généralement, si un pointeur `P` pointe sur une donnée de taille `T` , si l'on écrit `P+n`, on se réfère à l'adresse `P+(nxT)`.

Signalons au passage qu'il existe un opérateur spécialisé pour trouver la taille mémoire d'une donnée : il s'agit de `sizeof`.Par exemple,

```
sizeof(nb)
```

renverra 2 si `nb` est de type entier (2 octets).

RESUMONS-NOUS

Il est temps de nous résumer quelque peu. Les chaînes de caractères, sont accessibles de deux manières : soit comme un tableau, soit comme par pointeur.

Si vous la déclarez comme un tableau, par exemple par

```
char phrase[20];
```

la place de la chaîne est réservée, et le nom du tableau contient l'adresse de la chaîne; cette adresse ne peut pas être modifiée car les adresses de tableaux sont des constantes. Impossible donc d'assigner directement une nouvelle valeur à la chaîne; pour la modifier, il faudra utiliser `strcpy`, pour copier les caractères voulus un à un dans le tableau. Ces caractères sont accessibles par les indices du tableau (`phrase[0]`, `phrase[1]`, etc.). Souvenez-vous, le dernier octet d'une chaîne est toujours un 0, et le premier indice est toujours 0.

Si vous déclarez la chaîne par un pointeur, comme par exemple :

```
char *phrase;
```

vous pourrez la modifier directement, mais il faudra au préalable lui réserver une place à l'aide de `malloc`. Dans ce cas, le premier caractère de la chaîne sera `*phrase`, le second `*(phrase+1)`, c'est à dire un octet plus loin que le premier, et ainsi de suite.

A tout instant vous pouvez passer d'une notation à l'autre.

LES FONCTIONS LES PLUS UTILES

Turbo C comporte de très nombreuses fonctions qui manipulent les chaînes. Nous n'allons pas les décrire toutes ici, référez-vous à votre manuel Turbo pour cela. Nous nous contenterons de donner celles qui sont le plus fréquemment utilisées. Toutes ces fonctions se trouvent dans la librairie `<string.h>`. Si vous voulez vous servir de l'une ou plusieurs d'entre elles, vous devez inclure cette librairie dans votre programme par l'instruction :

```
#include  <string.h>;
```

Dans les syntaxes ci-dessous, `chaine`, `chaine1` ou `chaine2` représentent les adresses des chaînes de caractères; ces dernières peuvent avoir été déclarées comme des tableaux ou par des pointeurs. La ou les lignes de déclaration indiquent les types des opérandes éventuels.

strcat

Syntaxe:

```
chaine1 = strcat(chaine1,chaine2);
```

Ajoute une chaîne à la fin d'une autre : `chaine2` est ajoutée à `chaine1`. La longueur de la chaîne résultante est la somme des longueurs des chaînes initiales.

Exemple : Ajouter un espace à la fin d'un mot.

```
mot = strcat(mot," ");
```

strchr

Syntaxe:

```
char car, *lecar;
lecar = strchr(chaine1,car);
```

Cherche la première occurence du caractère `car` dans une chaîne, et renvoie un pointeur sur le caractère trouvé, ou NULL si le caractère n'existe pas. `lecar` est un pointeur sur un caractère.

Exemple :

```
char *lettre;
lettre = strchr("Bonjour!",'j');
```

Remarquez que cette fonction ne renvoie pas la position du caractère à partir du début de la chaîne, mais bien un pointeur sur ce caractère.

strcmp

Syntaxe :

```
int  code;
code = strcmp(chaine1, chaine2);
```

Compare `chaine1` et `chaine2`. Renvoie un entier indiquant le résultat de la comparaison :

si `chaine1 = chaine2`, `code=0`;

si `chaine1>chaine2`, `code` est positif;

si `chaine1<chaine2`, `code` est négatif.

La comparaison se fait sur les codes ASCII des caractères composant les chaînes, et tient compte des majuscules et des minuscules.

strlcmp

Identique à `strcmp`, mais ignore majuscules et minuscules.

strlen

Syntaxe :

```
int  longueur;
longueur = strlen(chaine);
```

Renvoie un entier non signé contenant la longueur de `chaine`, sans compter le caractère nul qui la termine.

strlwr

Syntaxe :

```
chaine = strlwr(chaine)
```

Toutes les majuscules de `chaine` sont converties en minuscules. La fonction retourne donc une chaîne composée uniquement de minuscules;

strset

Syntaxe :

```
char car;
chaine = strset(chaine,car);
```

Remplace tous les caractères de chaîne par `car`. Peut servir par exemple à remplir une chaîne d'espaces pour l'initialiser.

strnset

Syntaxe :

```
char  car;
int nombre;
chaine = strnset(chaine, car, nombre);
```

Semblable à la précédente, elle ne change que les `nombre` premiers caractères de `chaine`, en les remplaçant par `car`.

strstr

Syntaxe :

```
char  *lettre;
lettre = strstr(chaine1, chaine2);
```

Recherche la première occurence de la sous-chaîne `chaine1` à l'intérieur de `chaine2`, et renvoie un pointeur sur le caractère où elle commence dans `chaine2`. Si la sous-chaîne n'est pas trouvée, `strstr` renvoie NULL.

strupr

Syntaxe :

```
chaine = strupr(chaine)
```

Toutes les minuscules de `chaine` sont converties en majuscules.La fonction retourne donc une chaîne composée uniquement de majuscules.

CHAPITRE 7

LES DIRECTIVES ET LE PREPROCESSEUR

LE PREPROCESSEUR

Les éléments que nous venons d'examiner font partie intégrante du langage C. Nous allons voir maintenant comment profiter de toute la richesse apportée par un programme annexe du compilateur, le préprocesseur .

Le principe du préprocesseur est le suivant : comme son nom l'indique, il prend en charge un programme avant le compilateur, et transforme le code source en un code source composé uniquement des éléments standards du langage C.

Reprenons notre exemple HELLO.C :

```
main()
{
  printf("hello, world !\n");
}
```

Le compilateur C en lui même ne connaît pas la fonction `printf`. Ce qui peut être gênant, puisque c'est la fonction que nous utiliserons le plus souvent pour afficher des résultats !

Le travail du préprocesseur est de traiter cette fonction inconnue et de procurer une portion de code source que le compilateur C puisse traiter.

Les fonctions comme `printf` font partie de la "librairie standard". Cette librairie procure des fonctions très variées (environ 300, regroupées en une quinzaine de thèmes!). L'exemple de la fonction `printf` est particulièrement clair à ce sujet : sa déclaration véritable se trouve dans le fichier STDIO.H, ainsi que les données nécessaires à son intégration dans un programme C. Le préprocesseur est à même d'indiquer au compilateur comment inclure `printf` dans le programme.

Nous ne détaillerons pas ce sujet plus loin, car il faudrait faire la distinction entre les macros (qui sont transformées en code source et donc compilées), les déclarations (qui indiquent uniquement au compilateur où il peut trouver le code déjà compilé, de façon à l'indiquer lui-même au linker!) et les définitions (qui comportent le code en entier).

L'avantage de passer par un préprocesseur est le suivant : il permet d'enrichir le langage sans changer le compilateur. C'est très important car la mise au point d'un compilateur est compliquée et beaucoup plus longue que celle d'un préprocesseur. De plus, il facilite la portabilité de ces mêmes compilateurs entre

différentes machines, pour les mêmes raisons. Si l'on dispose d'un compilateur C standard sur une machine, il suffit de reprogrammer un morceau de préprocesseur pour lui faire adopter le même fonctionnement qu'un autre compilateur, sans autre forme de procès !

LES DIRECTIVES

D'un point de vue pratique, le préprocesseur accepte deux "directives", que nous pouvons utiliser dans les programmes. Ces directives ressemblent à des instructions mais n'en sont pas : elles ne seront jamais connues du compilateur car le préprocesseur les remplacera par du code.

Nous disposons des directives suivantes :

```
#include "nom de fichier personnel"

#include <nom de fichier standard>

#define CONSTANTE texte de la constante
```

La directive #include "nom de fichier"

Le rôle de cette directive, toujours placée en début de programme (comme les deux autres directives d'ailleurs) est d'inclure un fichier contenant un fragment de programme dans le votre programme lui-même. Supposons par exemple que vous ayez placé toutes vos fonctions d'affichage dans un fichier source appelé "AFFICHE.C". Ce fichier ne comporte pas de fonction main, et n'est donc ni exécutable ni compilable, mais vous pouvez l'inclure dans le texte de votre programme. Cela rendra celui-ci plus compact et sans doute plus clair. De plus, vous pourrez réutiliser ce même fichier AFFICHE.C dans un autre programme, sans devoir réécrire les fonctions d'affichage communes.

Notez que le compilateur, compile finalement l'ensemble de toutes les fonctions comme si elles provenaient d'un seul fichier. Cette directive est uniquement pratique pour le programmeur, elle ne modifie absolument rien à la compilation. Nous verrons ultérieurement comment effectivement découper un programme en modules compilés séparément.

La directive #include <fichier standard>

Cette directive a exactement le même effet que la précédente, à une nuance près toutefois : au lieu de chercher les fichier désignés dans le répertoire actuel (celui où se trouve donc votre programme), elle va les chercher dans le répertoire des fichiers inclus "standard". Si vous avez suivi nos instructions de configuration et d'installation, tous les fichiers .H de Turbo C se situent dans le répertoire "\C_INCL" ou "C\C_INCL" pour un disque dur, et ce répertoire a été indiqué à Turbo C grace au menu Options.

La directive ira dans ce cas chercher les fichiers indiqués entre "<" et ">" dans ce répertoire. C'est encore une fois uniquement pratique, cela évite de mélanger des fichiers qui n'ont pas grand-chose à voir entre eux. De plus, si vous vous procurez des librairies de fonctions toutes faites, vous pourrez les placer dans ce répertoire pour, toujours dans le même esprit, ne pas les mélanger pas avec vos propres sources.

Avec le compilateur Turbo C, la recherche des fichiers standard ne pose pas de problème. Vous avez pu remarquer que HELLO.C se compilait sans problème sans la moindre directive `#include`. Théoriquement, le programme devrait avoir l'aspect suivant :

```
#include <stdio.h>
main()
{
 printf("hello, world !\n");
}
```

Mais la plupart des compilateurs C incluent un préprocesseur capable de procéder implicitement à un `#include` pour les fonctions de la librairie standard. Toutefois, certaines fonctions nécessitent un `#include` en tête du programme. Nous les placerons lorsque le cas se présentera. Vous saurez dans tous les cas ce que signifie cette directive.

La directive #define CONSTANTE texte

Cette directive est pour sa part plus tournée vers la programmation. Elle permet de définir des constantes qui seront remplacées, dans le reste du programme, par le texte suivant le nom de la constante sur la même ligne. En voici un exemple concret :

```
#define PI 3.1415927

main()
{
  float rayon=23.0;
  float surface, circonference;
  surface = PI * rayon * rayon;
  circonference = 2 * PI * rayon;
}
```

Ce programme sera exactement identique au suivant :

```
main()
{
  float rayon=23.0;
  float surface, circonference;
  surface = 3.1415927 * rayon * rayon;
  circonference = 2 * 3.1415927 * rayon;
}
```

Ce dernier est tout de même plus fatigant à taper (et à lire), il faut le reconnaître. D'autre part, l'utilisation d'une constante permet de modifier facilement un programme comportant plusieurs fois l'utilisation d'une même valeur sujette à un éventuel changement. Si par exemple demain le continuum espace-temps est bouleversé et qu'à cette occasion la valeur de Pi est modifiée, il vous sera facile de remettre à jour vos programmes et de les recompiler en un clin d'œil.

Dans une optique tout de même plus réaliste, les constantes sont utilisées très souvent, soit pour les valeurs mathématiques ou arithmétiques fréquentes dans les programmes (par exemple un module calculant les intégrales pourra utiliser une constante E au lieu de 2,71828), soit pour les limites de tableaux que l'on envisage d'agrandir ou de diminuer. Par exemple, supposons que dans votre programme de calculs matriciels, vous ayez systématiquement déclaré des tableaux de 10 par 10, ceci à cause de la taille mémoire dont vous disposez. Si vous doublez votre mémoire à l'aide d'une extension et que vous souhaitez augmenter la puissance de traitement de vos fonctions, il va falloir changer toutes les déclarations et tous les indices de boucles ou tests se référant à ces valeurs. Quelle partie de plaisir ! Mais heureusement, vous aurez pris soin de définir deux constantes au début de votre programme, par exemple :

```
#define MAT_X_MAX 10
#define MAT_Y_MAX 10
```

Dans ce cas, il vous suffit de changer les valeurs de `MAT_X_MAX` et `MAT_Y_MAX` et de recompiler. Rapide, non ?

QUELQUES REMARQUES

Tout d'abord, la convention veut que l'on écrive les constantes définies par `#define` avec des MAJUSCULES. De cette façon, on les reconnaît immédiatement au sein du programme. N'oubliez pas que les compilateurs C différencient les majuscules et les minuscules. Vous avez par exemple le droit de définir une constante MAIN.

Ensuite, vous avez peut-être remarqué que les définitions de constantes ne se terminent pas par un point-virgule. Rappelons en effet qu'une définition est une DIRECTIVE, et n'a rien à voir avec une INSTRUCTION du langage. Le préprocesseur se contente de remplacer par exemple PI dans tout le programme par le texte qui suit `#define PI` sur la même ligne. Si vous avez placé un point-virgule, il recopiera le point-virgule. Inutile de dire que cela peut provoquer de magnifiques erreurs de compilation si ce point- virgule se place en plein milieu d'une expression ! C'est aussi pour cette raison qu'on écrit les constantes en majuscules, afin de ne pas oublier qu'elles sont remplacées par un texte.

Ne placez donc jamais de point-virgule dans une directive `# define`.

Enfin, il est d'usage courant de regrouper toutes les définitions de constantes dans un fichier séparé. Ce fichier (par exemple A_MOI.H), vous pourrez le placer soit dans votre répertoire au milieu de vos sources, soit dans le répertoire

des fichiers standards. Nous conseillons la seconde solution. De plus, le mieux est de nommer le fichier ainsi constitué avec une extension .H, pour le différencier des fichiers .C, qui seront de préférence des programmes compilables à part entière. Une fois votre fichier créé, vous l'enrichirez au fur et à mesure de vos progrès en C, jusqu'à ne plus pouvoir vous en passer. Il sera alors bien à l'abri dans le répertoire C_INCL, et il vous suffira de placer systématiquement une directive `#include <a_moi.h>` au début de vos programmes pour en profiter.

Enfin, pour l'anecdote, vous pouvez faire des choses extraordinaires avec `#define`. Regardez le programme suivant :

```
#define begin {
#define end }
#define write printf
#define program main()
program
begin
   write("hello world !\n");
end
```

Cela ne vous évoque rien ? C'est vrai que le Pascal est plus joli que le C, dans bien des cas. Si l'on excepte les quatre #define (qui pourraient se trouver, nous venons de l'envisager, dans un fichier A_MOI.H), c'est pratiquement du Pascal standard !

Toutefois, pour aussi tentant que cela puisse paraître, nous vous déconseillons franchement ce genre de manipulation.

D'une part votre programme ne sera plus lisible que par vous, et d'autre part vous risquez surtout très vite ne plus vous retrouver dans le C proprement dit ! Vous courrez le risque de limiter sérieusement votre progression en vous accrochant à des habitudes qu'aucun livre sur le C n'utilisera. Et si vous voulez progresser, autant le faire avec le langage courant !

CHAPITRE 8

VOCABULAIRE

LA SYNTAXE DE C

Le langage C, comme tout langage informatique, supporte extrêmement mal les approximations.

En clair, si le programme contient "prntf", jamais le compilateur n'ira supposer qu'il s'agit en réalité de "printf", bien que cela puisse nous sembler évident, à nous autres humains ! En conséquence, le développeur doit porter son attention sur le texte tapé et respecter avant tout ce que l'on appelle la syntaxe.

La syntaxe d'un langage est la définition précise et absolue de tout ce qui permet de le programmer : ses éléments, l'association de ces éléments, leur ordre d'apparition, leur nom exact, etc...

Pascal est l'exemple même d'un langage volontairement pointilleux sur le respect de la syntaxe. Son but était de forcer les étudiants à concevoir la majorité de leur travaux (y compris le programme lui-même) sur papier avant de se mettre au clavier, afin de limiter les fautes de frappe et les erreurs d'étourderies.

En langage C, c'est exactement l'inverse !

En effet, si Pascal a été créé pour faciliter l'enseignement de la grosse informatique aux étudiants (Basic s'avérant vite limité), C a été inventé pour faciliter la programmation d'applications de "bas niveau" (au sens très proche de la machine) par des spécialistes déjà compétents et expérimentés. Le but n'étant pas vraiment le même, les moyens pour l'atteindre n'en sont pas les mêmes non plus.

Ainsi, C possède-t-il une syntaxe extrêmement souple. Il est possible, par exemple, de mélanger des types, de modifier la structure de certaines données, de programmer vingt instructions en une seule, etc.

Par conséquent, un programme C peut être compact si on le désire, mais il peut devenir illisible si l'on ne prend pas garde.

Quoi qu'il en soit, certaines règles doivent dans tous les cas être respéctées. Ce chapitre les expose en détail.

IDENTIFICATEURS ET MOTS CLES

Le langage C manipule essentiellement des identificateurs, qui sont des noms donnés par le programmeur à certaines entités (types de données, variables, fonctions...) et des mots-clés qui sont en fait les identificateurs du langage C lu-même.

Les règles de base qui indiquent régissent la constitution des identificateurs et des mots-clés sont exactement les mêmes. Mais un identificateur ne peut pas porter le même nom qu'un mot-clé.

Les mots clés

Les mots-clés sont réservés: il est interdit de créer des fonctions, variables ou types portant le nom d'un de ces mots.

Le langage C "standard" comporte un petit nombre de mots-clés, mais tout compilateur en ajoute systématiquement quelques-uns, de façon à implémenter des fonctions de librairie utiles et/ou indispensables. Turbo C ne faillit pas à la règle et offre divers mots-clés pour implémenter six modèles de mémoire et un accès facile aux interruptions. De même, Turbo C inclut les mots-clés du récent standard ANSI.

Les mots spécifiques à Turbo C sont donc "en dehors" de tout standard et ne doivent pas être utilisés si le programme doit être transportable. Par contre les mots-clés ANSI se répandent progressivement et l'on peut choisir de les utiliser.

C Standard K&R			Turbo C		C ANSI
auto	(entry)	return	asm	_cs	const
break	extern	short	cdecl	_ds	enum
case	float	sizeof	far	_es	signed
char	for	static	huge	_ss	void
continue	goto	struct	interrupt		volatile
default	if	switch	near		
do	int	typedef	pascal		
double	long	union			
else	register	unsigned			
		while			

Note : le mot-clé `entry` est réservé dans la norme K & R, mais la plupart des compilateurs ne l'utilisent pas, et le standard ANSI l'a tout simplement éliminé. Il devait permettre une extension du langage, mais il n'a jamais réellement été employé. Turbo C ne le reconnaît pas , mais par précaution, si l'on réalise une application portable, on évitera de l'employer comme identificateur, car certains compilateurs le reconnaissent encore.

Ce tableau exprime peut-être plus clairement qu'un long discours la différence entre C et les autres langages : la liste des mots réservés de Pascal est beaucoup plus grande; quant à Basic, n'en parlons pas, il est difficile de les retenir tous !

Si les mots-clés sont si peu nombreux, c'est que, comme nous l'avons déjà souligné, C est un langage maigre ! La quasi-totalité de ses fonctionnalités provient des fonctions de la librairie standard, qui procurent l'équivalent des fonctions ou instructions classiques d'autres langages. Rappelez vous que la librairie fournie par Borland procure plus de trois cents fonctions diverses directement utilisables dans les programmes C. Ces trois cents fonctions peuvent très bien être ignorées : les mots-clés du C procurent le minimum avec lequel on peut pratiquement tout faire.

Toutefois, il faut remarquer que le langage C, en standard, est destiné aux problèmes abstraits : il ne comporte aucune instruction ou fonction d'affichage, de gestion de fichiers ou de saisie au clavier ! Ces fonctionnalités sont réalisées par des fonctions de la librairie.

LES IDENTIFICATEURS, NORME K&R

Un identificateur est un mot composé de plusieurs caractères, qui répondent aux règles suivantes :

- Le premier caractère est obligatoirement une lettre (majuscule ou minuscule), ou bien le caractère souligné (_);

- Les caractères suivants sont soit des lettres, soit des chiffres, soit le souligné. Tout autre caractère est interdit, ou marque la fin de l'identificateur;

- Les majuscules et les minuscules ne sont pas identiques (`ToTo` est différent de `toto` et de `TOTO`);

- Les six premiers caractères sont les seuls pris en compte (`maison` et `maisonette` sont un seul et même identificateur).

LES IDENTIFICATEURS, TURBO C

Turbo C comporte les mêmes règles mais étend leur limites:

- Le premier caractère répond aux mêmes règles;

- Le dollar "$" est également un caractère valide;

- La différence majuscules-minuscules peut être supprimée (menu OPTION/LINKER/Case sensitive Link);

- Les 32 premiers caractères sont significatifs, mais on peut réduire ce nombre (menu OPTION/COMPILER/SOURCE);

Notez que pour rester standard, il vaut mieux éviter d'utiliser ces possibilités supplémentaires.

LES OPERATEURS

Le langage C comporte une liste impressionante d'opérateurs, dont certains, il faut bien l'admettre, sont relativement peu lisibles.

En fait, la quasi totalité des caractères accessibles au clavier a été utilisée pour représenter une opération C.

Vous trouverez ci-dessous la liste les opérateurs par ordre décroissant de priorité et par type d'opération.

OPERATEURS UNAIRES (un seul paramètre) :

Non logique : !

```
! exp
```

Renvoie l'inverse de la valeur logique de `exp` (`1` si `exp` est faux et 0 si `exp` est vrai).

Incrémentation : ++

```
++ variable
```

ou

```
variable++
```

Incrémente la valeur de `variable` suivant son type : si `variable` est de type numérique (`char`, `int`, `short`, etc.), cela ajoute 1 à `variable`. Si `variable` est un pointeur, le nombre ajouté dépend de la taille mémoire du type pointé (`sizeof (var)`).

Si l'opérateur est placé devant la variable, l'incrémentation a lieu avant son utilisation; s'il est placé après, elle a lieu après celle-ci.

Décrémentation : --

```
-- variable
```

ou

```
variable --
```

Identique à ++, mais opère une décrémentation (soustraction).

Inversion de signe : -

```
- exp
```

Renvoie la valeur opposée de `exp`.

Casting

```
(type) exp
```

Effectue la conversion de `exp` en type `type`.

Indirection : *

```
*ptr
```

Représente la valeur pointée en mémoire par `ptr`.

Adresse : &

```
& variable
```

Renvoie l'adresse de `variable`.

Taille : sizeof

```
sizeof (nom)
```

Renvoie le nombre d'octets occupé par le type ou la variable `nom`.

OPERATEURS BINAIRES (manipulation de bits)

NON binaire : ~

```
~ exp
```

Renvoie la valeur binaire `non exp` (tous les bits inversés).

Décalage à gauche : <<

```
<< exp
```

Renvoie la valeur binaire `exp` décalée à gauche d'un bit (multipliée par 2).

Décalage à droite : >>

```
>> exp
```

Renvoie la valeur binaire `exp` décalée à droite d'un bit (division entière par 2).

ET binaire :&

```
(exp1) & (exp2)
```

Renvoie la valeur du ET binaire entre `exp1` et `exp2` (ET bit à bit).

OU binaire : |

`(exp1) | (exp2)`

Renvoie la valeur du OU binaire entre `exp1` et `exp2` (OU bit à bit).

XOR binaire : ^

`(exp1) ^ (exp2)`

Renvoie la valeur du XOR (ou exclusif) binaire entre `exp1` et `exp2` (XOR bit à bit).

OPERATEURS ARITHMETIQUES

- `*` : multiplication
- `/` : division
- `%` : modulo (reste par divison entière)
- `+` : addition
- `-` : soustraction

OPERATEURS RELATIONNELS (expression conditionelles)

Test d'infériorité : <

`(exp1)<(exp2)`

Renvoie 1 si `exp1` est strictement inférieur à `exp2` (ou 0 sinon).

Test de supériorité : >

`(exp1)>(exp2)`

Renvoie 1 si `exp1` est strictement supérieur à `exp2` (ou 0 sinon).

Test d'infériorité ou égalité : <=

`(exp1)<=(exp2)`

Renvoie 1 si `exp1` est inférieur à `exp2` (ou 0 sinon).

Test de supériorité OU égalité : >=

`(exp1)>=(exp2)`

Renvoie 1 si `exp1` est inférieur à `exp2` (ou 0 sinon).

Test d'égalité : ==

```
(exp1)==(exp2)
```

Renvoie 1 si `exp1` est égal à `exp2` (attention au double signe=),ou 0 sinon.

Test de non-égalité : !=

```
(exp1)!=(exp2)
```

Renvoie 1 si `exp1` est différent de `exp2` (ou 0 sinon).

ET logique : &&

```
(exp1)&&(exp2)
```

Renvoie 1 si `exp1` ET `exp2` sont logiquement vrais, c'est-à-dire différents de 0 (ou 0 sinon).

OU logique : ||

```
(exp1)||(exp2)
```

Renvoie 1 si `exp1` OU `exp2` sont logiquement vrais, c'est-à-dire différents de 0 (ou 0 sinon).

OPERATEURS D'AFFECTATION

L'opérateur d'affectation est le signe =. Il signifie "ranger dans l'identificateur de gauche le résultat de l'expression de droite". Exemple :

```
circonference = 2*PI*rayon;
```

Mais le C, rusé et malin, donne certains raccourcis pour effectuer en même temps une opération arithmétique :

```
+= : destination = destination + expression

-= : destination = destination - expression

*= : destination = destination * expression

/= : destination = destination / expression

%= : destination = destination % expression

|= : destination = destination | expression

^= : destination = destination ^ expression

&= : destination = destination & expression

>>= : destination = destination >> expression

<<= : destination = destination << expression
```

Exemples :

`a += b` est équivalent à `a = a+b`

`registre &= masque` est équivalent à `registre = registre & masque`

`i *= i` est équivalent à `i = i*i`

OPERATEUR D'AFFECTATION CONDITIONELLE

L'opérateur "?" permet une écriture très concise.

```
(a?(b):(c))
```

Si condition `a` est vraie (différente de 0), cette expression vaut `b`,sinon `c`. Les parenthèses autour de `b` et `c` ne sont pas obligatoires, mais elles sont prudentes.

Exemple :

```
printf("%s",(a<b)?"a<b":"a>=b");
```

Cette seule ligne affiche "a<b" si `a` est réellement inférieur à b, et "a>=b" sinon.

SEPARATEUR D'EXPRESSIONS

La virgule ",", sépare des expressions regroupées dans une seule expression :

```
(exp1, exp2 ...)
```

Exemple :

```
(i++,j=i*i*i)
```

Cette expression prend la valeur de `j` après avoir incrémenté `i` et rangé son cube dans `j`.

EXPRESSIONS

En langage C, les expressions sont beaucoup plus puissantes que dans la plupart des autres langages.

La grande astuce est la suivante :

Toute expression possède intrinsèquement la valeur qu'elle calcule.

Exemples :

`(c=2)` prend la valeur "2", en plus de ranger 2 dans `c`. On peut assigner cette valeur à une variable, dans un test, dans une expression, etc...

`(a=(b=c))` donne la valeur de c à b, ce qui donne la valeur `c` à `b=c`, donc `a` prend également la valeur de `c`.

`i=(i==0)` donne à i la valeur 1 si elle valait 0 , et 0 sinon : dans le cas où `i` vaut 0, "`i==0`" est vrai donc vaut 1, et `i` prend ensuite cette valeur. Si `i` ne vaut pas 0, "`i==0`" est faux et `i` devient 0 !

On peut multiplier les exemples à l'infini. Une expression a aussi, indépendamment de sa valeur propre, une valeur logique: `faux` si elle vaut 0, `vrai` si elle est différente de zéro. Cette particularité remplace le type `Boolean` du Pascal.

D'autre part, on peut fabriquer une expression composée :

```
(expression, expression...)
```

L'opérateur "," est le séparateur d'expressions. Le tout doit être inclus entre parenthèses, et est évalué de la gauche vers la droite (expression par expression). Le tout est UNE expression, qui a la valeur de la dernière composante calculée !

Par définition, une expression n'est pas une instruction. Elle en devient une si on lui ajoute une assignation et un point-virgule. Toute assignation doit comporter ce que l'on appelle une "lvalue" ("l" pour "left" = gauche en anglais, "value" pour valeur), c'est-à-dire une entité pouvant recevoir et stocker le résultat de l'expression qui lui est assignée. Prenons l'exemple suivant :

```
k = (x * 2);
```

La variable `k` doit être du même type que `x * 2`, ou du moins d'un type compatible. Par exemple, si `k` est du type `double` et x de type `short`, l'assignation est possible (rappelons que le type `short` est semblable à `int` en Turbo C).

La grande force de C, c'est que cette assignation peut également être valide si `k` est de type char et `x` du type `short`, par exemple, à condition que `2 * x` soit dans les limites du type `char`, bien sûr. L'inverse est également vrai.

Par contre, attention aux assignations à base de pointeurs.

Vous devez bien assimiler le fait que le type pointeur est une sorte d'entier spécialisé. C ne vous interdira pas de modifier le contenu d'un pointeur. Par exemple, si `a` et `b` sont deux pointeurs vers des entiers, vous pouvez parfaitement procéder à l'assignation suivante :

```
a = b;
```

Cela ne provoquera rien de fâcheux. Les choses sont beaucoup plus délicates avec l'opérateur d'indirection :

```
*a = *b;
```

Théoriquement, cette assignation range le contenu entier pointé par `b` à l'endroit pointé par `a`. Ca fonctionne très bien... à moins que ni `a` ni `b` ne pointent sur une zone réservée au stockage d'un entier . En effet, un pointeur,

tant qu'on ne lui a pas réservé une zone mémoire et qu'on ne le fait pas pointer sur cette zone, pointe n'importe où. Ce peut être aussi bien vers une zone totalement inexploitée de la mémoire, auquel cas rien de grave ne transparaît, mais ce peut être aussi en plein au milieu du programme, auquel cas... gare au blocage!

La séquence suivante est par contre sans danger :

```
short  *pa,*pb; /* déclaration de deux pointeurs de short */
short  a, b;    /* déclaration de deux short */
pa = &a;        /* pa pointe sur la zone réservée par a */
pb = &b;        /* pb pointe sur la zone réservée par b */
*pa = *pb;       /* équivalent à a=b */
```

La plupart des erreurs graves ou subtiles dans les programmes en C proviennent de l'utilisation de pointeurs non initialisés. Turbo C indique ce genre d'anomalie avec des "Warning" (avertissements) lors de la compilation. Par exemple, si la séquence ci-dessus ne comporte pas les lignes `pa=&a` ou `pb=&b`, Turbo C vous le signalera par le message "il est possible que vous assayiez d'utiliser `pa` (ou `pb`) avant qu'ils ne soient initialisés", ou quelque chose d'approchant en anglais.

Notez bien, c'est là le drame et l'intérêt de la chose, que ce type d'anomalie peut être voulu, aussi le programme sera-t-il parfaitement exécutable. Un warning est une alerte, pour éveiller l'attention du programmeur sur un point litigieux, mais ce n'est pas une erreur de compilation.

Soyez vigilant sur les expressions : C est extrêmement souple et tolérant à ce niveau, et autorise pratiquement tout. En général, si Turbo C donne un Warning, c'est qu'il y a possibilité d'erreur. Vous aurez souvent des warnings sur des expressions.

Enfin, encore un point important, une assignation n'est pas forcément constituée par un opérateur "=". Vous disposez en effet de tous les opérateurs combinés comme "+=" ou "~=". Rappelons que cette notation est uniquement un raccourci, et qu'elle équivaut à une expression plus classique :

```
lvalue opérateur= expression;
```

équivaut à

```
lvalue = lvalue opérateur expression;
```

Par exemple, les notations suivantes sont équivalentes :

```
i = i+j
```

et

```
i+=j;
```

Utilisez des parenthèses. En effet, étant donné la souplesse autorisée par les expressions (les tests sont des expressions, une expression a la valeur du résultat qu'elle calcule, une assignation aussi, une expression peut être

composée, etc.), il n'est pas rare que l'oubli d'un niveau de parenthèses provoque de grosses anomalies. Notre conseil est le suivant :

Même si cela paraît inutile, placez des parenthèses autour de toute expression simple, de façon à séparer ses opérandes des autres intervenants dans le calcul.

ASSIGNATIONS MULTIPLES ET EXPRESSIONS COMPOSEES

Des précisions s'imposent concernant deux des originalités du langage C sur ses concurrents.

Expression composée

```
(expression , expression , ...)
```

Cette expression est exécutée en séquence, de l'expression simple la plus à gauche vers celle de droite. L'effet de chacune des expressions intervient donc avant la suivante. Ne l'oubliez pas, c'est très important. Si vous faites ceci :

```
for ( i=1; i<=10 ; (++i, a=++i) );
```

`i` sera incrémenté une première fois (par `++i`), puis une deuxième fois avant d'être assignée à `a` (`a=++i`). Cette dernière variable prendra donc les valeurs successives 3, 5, 7 et 9 (voir `for`).

Assignations séquentielles

```
lvalue opérateur = lvalue opérateur = ... opérateur expression;
```

Contrairement à l'expression composée qui est exécutée de gauche à droite, ce type d'expression est examiné de droite à gauche (sauf si la présence de parenthèses redéfinit cette ordre). L'expression finale de droite est calculée. Sa valeur est appliquée au dernier opérateur, et le résultat va dans la dernière lvalue. Cette lvalue étant assignée à une valeur, cela donne une nouvelle valeur qui est reportée à gauche, et ainsi de suite. Exemple :

```
a = (b = (c!=0));
```

Dans ce cas, si `c` est vrai (différent de zéro), `(c!=0)` vaut 1, donc `(b = 1)` est exécutée. Cette expression vaut elle-même 1, donc `a = 1` est exécuté.

Dans la plupart des cas similaires, l'omission de parenthèses rend les choses illisibles ou beaucoup plus claires, cela dépend ! Notez par exemple la différence de clarté ci-dessous :

```
a = (b = (c = (d == 0)));
```

```
a = b = c = d == 0;
```

Il s'agit exactement de la même expression. L'une est plus sûre, l'autre est plus claire. Au lecteur de choisir son style !

Notez qu'en plus des assignations, vous pouvez procéder à d'autres opérations, même s'il faut obligatoirement une assignation pour fabriquer une instruction :

```
a=((b*2)==(((c=3)+15)==30));
```

Cette expression est presque illisible mais légale : tout d'abord on procède à l'assignation `c=3`, qui range 3 dans la variable `c` et donne une valeur 3. On ajoute 15, ce qui donne 18. On compare à 30, ce qui donne 0 (=faux). On compare ensuite cela à `2 * b`, et le résutat de ce test est rangé dans `a`.

CHAPITRE 9

INSTRUCTIONS DE BASE

Les instructions du langage C sont réellement peu nombreuses. La plupart des actions sont réalisées par des fonctions de la librairie standard.

Par principe, le langage C est en effet basé sur l'utilisation de fonctions dans tous les domaines. C'est ce qui fait sa force. Cependant, il existe bien entendu un minimum de structures de contrôle, d'instructions de boucles, etc...

Ces instructions, qui ne sont donc absolument pas comparables aux autres éléments du langage, forment en quelque sorte le noyau du langage, car justement leur faible nombre les rend indispensables.

D'un point de vue formel, les mots-clés comme `for` ou `while` ne sont pas les seuls éléments de C qui puissent être considérés comme instructions. En effet, l'exemple suivant est aussi une instruction :

```
variable = 15;
```

Nous avons donc séparé logiquement dans ce chapitre les instructions du langage proprement dit, et celles qui sont des assemblages d'autres éléments.

INSTRUCTIONS SIMPLES ET COMPOSEES

D'autre part, on doit distinguer les instructions simples et les instructions composées. Les deux sont équivalentes : partout où l'on peut utiliser une instruction simple, on a le droit d'utiliser une instruction composée. On regroupe ces deux formes sous le terme général d'instruction, sans précision supplémentaire. D'ailleurs, une instruction composée n'est rien d'autre qu'une suite d'instructions simples (avec toutefois une nuance).

Dans la suite de ce chapitre, nous apellerons `instruction` (en général) l'une des choses suivantes :

- une `instruction simple;`

Exemple :

```
printf("Le C est formidable!\n");
```

- le" ;"

- le groupe :

```
{
Déclarations de types locaux
Déclarations de variables locales
Définitions de données locales
instruction
instruction
...
}
```

c'est-à-dire une `instruction composée` (SANS point-virgule).

Exemple :

```
for (i=1; i<10; i++)
{
  j = i * i;
  printf("i= %d, i*i = %d\n", i,j);
}
```

Notez dans ce cas l'absence de point-virgule. En réalité, une instruction simple se termine toujours par un point-virgule, mais pas une instruction composée.

Enfin, l'intérêt de l'instruction composée est qu'on peut lui incorporer des définitions et déclarations locales, ce que Pascal ne permet pas. Vous pouvez par exemple écrire ceci :

```
for (i=1; i<10; i++)
{
  short c;
  c = i * i;
}
```

Dans l'instruction (composée) qui est appliquée au `for`, la variable `c` est locale; elle est créée puis détruite et à chaque tour de boucle.

INSTRUCTION D'AFFECTATION

Syntaxe:

```
identificateur opérateur expression ;
```

La zone de stockage (mémoire) représentée par l'identificateur reçoit le résultat de l'expression traitée par l'opérateur. Notez le point-virgule final, qui appartient à l'instruction à part entière. Cette forme est une instruction simple.

Exemple :

```
i += 1;
```

Ce qui est équivalent à :

```
i = i + 1
```

INSTRUCTIONS CONDITIONNELLES

Comme tout langage, Turbo C comporte des instructions conditionnelles. Elles sont au nombre deux : `if` et `switch`.

L'instruction if

Syntaxe:

```
if ( expression ) instruction
```

ou

```
if ( expression ) instruction else instruction
```

Ces deux formes sont analogues à ce que l'on trouve dans les autres langages comme BASIC ou Pascal. La nouveauté vient de l'interprétation du test.

Le langage C ne comporte pas de type booléen. Il réagit comme Basic : pour lui, toute expression qui à pour valeur 0 est assimilée à la valeur logique "faux". Et, bien sûr, toute expression différente de 0 représente "vrai".

Exemple:

```
if (a==b)
  puts("a égale b");
else
  puts("a <> b");
```

Comme pour Basic, ce principe permet de procéder à des tests extraordinairement souples, mais c'est aussi une source possible d'erreur car les opérateurs d'affectation ressemblent étrangement à ceux de condition, et les deux peuvent être admis par le compilateur :

```
if (a=b) & (c=d) instruction
```

est tout aussi valide que :

```
if (a==b) && (c==d) instruction
```

Mais l'effet est totalement différent, et le principe également. Reportez-vous au chapitre traitant des expressions pour avoir plus de précisions.

D'autre part, l'instruction qui suit l'expression de test doit se terminer par un point-virgule, ou bien être une instruction composée.

IMPORTANT : PAS DE "THEN" !

Si vous avez tendance, comme la majorité des programmeurs Pascal et BASIC, à écrire un "then" dès qu'il est question d'if, ce n'est pas grave. Vous pouvez ignorer cet aspect purement linguistique en incorporant systématiquement les lignes suivantes au début des programmes :

```
#define then
#define THEN
```

Par la suite, tout `then` ou `THEN` sera remplacé (à la compilation) par ... rien du tout. L'erreur de syntaxe disparaît, vous n'avez plus à vous en préoccuper.

Remarquez aussi une autre différence importante par rapport au Pascal: `else` peut être considéré comme une instruction à part entière. Cela signifie que l'instruction qui suit le `if` et qui précède le `else` doit comporter un point-virgule. La encore, attention aux habitudes et aux réflexes !

Un `else` se rapporte toujours au `if` précédent sans `else` associé, à moins que celui-ci ne soit pas dans le même bloc (auquel cas il y a une erreur de conception et de compilation...)

On peut réaliser des tests mutuellement exclusifs de façon très simple en enchaînant des `else` avec des `else if` :

```
if (exp1)
 instruction1
else if (exp2)
 instruction2
else if (exp3)
 instruction3
...
else
  instruction_finale
```

Cette forme de structure peut également être obtenue par l'instruction `switch`.

L'instruction switch

Syntaxe:

```
switch ( expression )
{
case constante :
  instruction
  ...
  [ break; ]
case constante :
  instruction
  ...
  [ break; ]
...
[default :
  instruction
  ...
]
}
```

Ci-dessus, les "[" et "]" représentent des options.

La valeur de `expression` est tout d'abord calculée. Puis les `case` *`constante`* sont examinés : si l'un correspond (la *`constante`* a la valeur de l'`expression`), les instructions placées au-delà du `":"` sont exécutées. Si aucune constante ne correspond, `default` est pris par défaut, s'il est présent, et ses instructions sont exécutées.

Si une séquence d'instructions est exécutée, le `switch` continue le test avec les `case constante` suivants, à moins que la séquence d'instructions ne se termine par `break;`, qui met fin au travail (sortie du bloc).

Exemple :

```
switch ( moncar)
{
  case 'a' :
        puts("Vous avez tapé (A)nnulation …");
        annulation();
        break;
  case 'f' :
        puts("Vous avez tapé (F)in…");
        fin();
        break;
  default :
        puts("Tapez A pour annuler, F pour fin.")
}
```

Remarques :

1. Il est plus que conseillé d'utiliser systématiquement `break;` à la fin de chaque séquence d'instructions. Sinon, le test se poursuivra (généralement inutilement). Il est dangereux de procéder autrement, car la valeur peut parfois correspondre à plusieurs des constantes, et chaque séquence d'instruction serait alors exécutée.

2. Les instructions de chaque `case` *`constante`* ne sont pas des instructions composées. Il n'y a pas d'accolade. Par contre il faut bien entendu un point-virgule à la fin de chacune (`instruction` comporte implicitement un point-virgule), et si possible une instruction `break;` pour marquer la fin de la séquence.

3. Le test effectué pour chaque `case` est `if (expression == constante)`. Il s'agit d'un test d'égalité absolue. On évitera de procéder à un tel test sur des nombres réels.

4. Attention aux tests de type booléens : vous pouvez tester si une expression est fausse, pas si elle est vraie ! Exemple :

```
#define VRAI 1;
#define FAUX 0;
switch ( expression )
{
  VRAI : blabla;
         break;
  FAUX : blabla;
         break;
}
```

Ceci ne fonctionne que si `expression` ne prend que les valeurs 0 ou 1. Il vaut mieux utiliser ceci :

```
#define FAUX 0;
switch ( expression )
{
  FAUX : blabla;
         break;
  default : blabla;
}
```

Cette structure fonctionne parfaitement : tous les cas qui ne sont pas FAUX (`expression == 0`) passeront par `default`.

LES INSTRUCTIONS DE BOUCLE

Classiques elles aussi, elles n'en sont pas moins, surtout pour le `for`, particulièrement puissantes en C.

L'instruction while

```
Syntaxe:
while ( expression ) instruction
```

`instruction` (simple ou composée) est exécutée tant que l'`expression` a une valeur vraie (différente de 0). Si l'`expression` vaut 0 dès l'entrée, `instruction` n'est jamais effectuée.

Exemple :

```
i = 1;
while (i*i<200)
{
  printf("i=%d; i*i=%d\n",i,i*i);
  i +=1;
}
```

La boucle s'effectue tant que le carré de `i` est inférieur à 200. Notez l'incrémentation de `i` par l'opérateur += (equivalent à `i=i+1` ou encore à `i++`).

Attention, `instruction` doit modifier le résultat de `expression`, sinon on risque de ne jamais sortir de la boucle; essayez ceci :

```
main()
{
  int i=j=1;
  while (i==j)
        printf(" Pour l'éternité…\n");
}
```

Cette instruction est en quelque sorte la plus belle de toutes, car avec elle, on peut simuler tout ce qui existe, du `if` au `for`, sans oublier les `goto` et `do-while` (voulez-vous tenter l'exercice? C'est assez simple, et ce n'est pas inintéressant).

L'Instruction do..while

Syntaxe :

```
do instruction while (expression);
```

`instruction` est exécutée jusqu'à ce que `expression` prenne la valeur 0 (faux). Notez le point-virgule final : il est obligatoire pour terminer le `do while` car le dernier membre (`expression`) n'est pas une instruction et ne comporte donc pas de point virgule. Les parenthèses sont obligatoires autour de `expression`.

Exemple :

```
main()
{
  int nombre, continuer = 0;
  d o
  {
        printf("Tapez un nombre inférieur à 10\n");
        scanf("%d",&nombre);
        continuer = (nombre<10);
  }
  while (continuer);
}
```

La seule difficulté est la ligne `continuer = (nombre<10);` elle évalue l'expression `nombre <10`, et en range la valeur (o si faux, 1 si vrai) dans `continuer`.

Remarquez qu'avec `while...do instruction` est au moins exécutée une fois, même si `expression` est fausse en entrée.

L'instruction for

Syntaxe :

```
for ( [ exp1 ] ; [ exp2 ] ; [ exp3 ] ) instruction
```

L'exécution de cette boucle a lieu en plusieurs étapes :

- `exp1` est calculée.
- `exp2` est calculée.
- Si `exp2` est vraie :
  - `instruction` est exécutée.
  - `exp3` est calculée
  - retour en 1.
- sinon fin.

Essayez cet exemple :

```
main()
{
  char carac;
  for (carac='a';carac <='z';carac++)
  printf("%c",carac);
}
```

Remarques :

D'un point de vue formel, on considère que l'instruction `for` possède la forme suivante :

```
for (initialisation;test pour continuer;mise à jour) instruction
```

Il est préférable d'utiliser une variable de contrôle pour la boucle, afin d'obtenir quelque chose du style :

```
for ( var=debut ; var<=fin ; var++) instruction
```

La boucle `instruction` sera exécutée pour `var` prenant les valeurs de `debut` à `fin` (comprises).

Notez cependant que l'initialisation, le test de continuation et la mise à jour peuvent se passer de variable de contrôle, à condition que `instruction` permette à l'`expression test` de prendre la valeur 0 !

ENFIN, ultime remarque, l'instruction `for` est connue pour être l'une des plus fantastiques du langage C. Il est vrai que son aspect "tout en un" est assez frappant. Mais n'en abusez pas. Certains prétendent qu'on peut faire des choses extraordinaires en une seule boucle `for`. C'est vrai. Mais on réussit surtout, à ce jeu là, à faire des choses extraordinairement illisibles.

Reportez-vous spécialement au chapitre des expressions, où vous verrez comment une expression peut être composée de plusieurs expressions, un peu comme une instruction peut en contenir plusieurs.

INSTRUCTIONS DE CONTROLE

L'instruction return

Syntaxe :

```
return ( expression );
```

Utilisée dans le bloc d'une fonction, cette instruction sort de la fonction et renvoie le résultat de `expression`. Ceci ne fonctionne que si l'identificateur de la fonction est d'un type compatible avec l'expression renvoyée (sous peine d'une erreur à l'exécution).

`return` est optionnel. On peut l'omettre si la fonction n'est pas destinée à renvoyer de résultat (par exemple, une fonction pour effacer l'écran). Dans ce cas, on déclarera de préférence la fonction comme type `void` (voir le chapitre sur les fonctions et leur déclaration).

Instruction Continue

Syntaxe :

```
continue;
```

Cette instruction bizarre a un effet différent suivant la structure qui l'emploie :

- Dans une instruction `for`, où qu'elle se trouve, elle renvoie directement à l'étape de la mise à jour (expression 3).

- Dans un `while` ou `do-while`, elle renvoie directement à l'évaluation de l'expression de test.

L'intérêt de `continue` apparaît surtout dans les structures `while` et `do-while`, lorsqu'une condition qui devrait générer une sortie immédiate se réalise.

Dans un tel cas, les mauvais programmeurs utilisent un `goto`. Fi! Les plus minutieux, structurent le tout dans un `if then else` géant avec un drapeau d'alerte. Cela marche, et c'est déjà plus propre. Nous préférons `continue`, qui en fait revient au même mais sans utiliser de `if else`. Voici une exemple, avec les trois solutions.

Exemple théorique :

```
while (dites != 33)
{
  if condition
  {
        truc;
        il faudrait sortir du while sans changer la valeur de "dites";
  }
  autres trucs...
 }
```

Il y a plusieurs manières de réaliser cela, en employant différentes instructions :

- Solution avec `goto` :

```
while (dites != 33)
{
  if condition
  {
        truc;
        goto ouhlala;
  }
  autres trucs...
}
ouhlala:....
```

- Solution avec `if` :

```
drapeau = 0;
while ( (dites != 33) && (drapeau == 0) )
{
  if condition
  {
        truc;
        drapeau = 1;
  }
  else
  {
        autres trucs...
  }
}
```

- Solution encore plus claire :

```
drapeau = 0;
while ( (dites != 33) && (drapeau == 0) )
{
  if condition
  {
        truc;
        drapeau = 1;
        continue;
  }
  autres trucs...
}
```

Toutefois, on peut préférer la deuxième solution; c'est essentiellement une question de goût. Mais si le bloc comporte de nombreuses imbrications, il est épineux d'en rajouter une, et surtout en imposant un `else` on prend le risque d'oublier qu'il existe ! Avec `continue`, on sait que le `while` ou `do while` immédiatement au dessus est terminé, et la clarté du programme n'en souffre donc pas.

Notez cependant que `continue` équivaut ici à l'utilisation de `goto` (mais le `else` également). Ne vous y fiez pas : c'est notre exemple qui est favorable, car dans un tel cas, l'utilisation de `goto` n'est en réalité pas trop grave. Elle fait même gagner une variable. Mais ne soyons pas trop mesquins !

L'instruction break

Syntaxe :

```
break;
```

Cette instruction, utilisable dans `switch`, `while`, `for` et `do while`, provoque une sortie immédiate de la structure.

Avec break, la solution ultime à notre petit problème donne ceci :

Solution parfaite :

```
while (dites != 33)
{
  if condition
  {
        truc;
        break;
  }
  autres trucs
...
}
```

La plus propre, la plus économique, et la plus rapide !

Malheureusement, la plupart des programmeurs hésitent à utiliser `break` pour sortir d'une boucle sans repasser par le test. D'un strict point de vue structuration, cela se justifie : au moins, en sortant par `continue` avec un drapeau, nous sommes certains d'être sortis volontairement de la boucle. Mais en réalité, nous en sommes tout aussi certains avec `break` !

Comme pour la majorité de toutes ces instructions, il appartient au lecteur d'adopter la solution qui lui plaît, sachant que les quatre solutions de notre petit problème sont rigoureusement identiques au niveau du fonctionnement (evitez `goto`, tout de même...).

L'instruction exit()

Syntaxe:

```
int code;

exit();
```

ou

```
exit(code);
```

Stoppe l'exécution du programme et retourne sous MS-DOS. `code` est facultatif et représente le code de retour passé au DOS. `code` peut ainsi être testé par une commande `IF ERRORLEVEL code` dans un fichier .BAT.

Un programme sous MS-DOS renvoie habituellement 0 si son exécution s'est correctement déroulée, et une valeur différente s'il s'est terminé par une erreur.

`exit(code)` est utile pour stopper l'exécution lorsqu'une erreur grave est détectée, et ainsi signaler au DOS ou au programme BAT appelant que telle erreur est survenue. Si `exit ()` est présent, la valeur 0 est renvoyée; de même si le programme se termine sans rencontrer `exit()`.

L'instruction goto

Syntaxe :

```
goto etiquette;
etiquette:instruction
```

Permet un branchement à une étiquette. Complètement à proscrire lorsqu'on veut faire une programmation structurée. Au mieux, goto permet d'embrouiller facilement les esprits les plus clairs. Oubliez cette instruction.

CHAPITRE 10

STRUCTURATION

QU'EST-CE QUE LA PROGRAMMATION STRUCTUREE ?

Une fois n'est pas coutume, dans cette partie du chapitre nous n'utiliserons pas le clavier !

D'ailleurs, un peu de repos ne fera pas de mal, n'est-ce pas ? Pour un moment, oublions un peu les pointeurs, les chaînes de caractères ou les fonctions. Nous allons parler structuration.

Pourquoi parler de programmation structurée ?

Nous avons lu tellement de contre-vérités sur cette méthode de travail, entendu tant d'absurdités, vu tellement de non-sens, depuis de nombreuses années, qu'il est temps de faire le point.

Pour notre part, nous avons découvert le véritable sens de la programmation structurée grâce à un professeur de faculté. Sa méthode était de nous forcer à analyser un programme ... sans utiliser de langage, et donc sans `goto` (puisqu'à l'époque, à moins de 35000 francs, on ne pouvait pas programmer en autre chose que Basic ou langage machine) ! La révélation n'a pas frappé tout le monde, mais ceux qui ont un jour compris le "pourquoi" et le "comment" de cette façon de travailler font aujourd'hui les plus belles choses en informatique.

Y a-t-il quelque part quelque chose en plus, une théorie spectaculaire et mystérieuse, une notion difficilement accessible au commun des mortels ?

Pas du tout. C'est là que cela devient extraordinaire : la programmation structurée procède d'une démarche naturelle qui devrait être innée.

Tout d'abord, qu'est-ce que la programmation structurée ?

Certains prétendent que Basic n'est pas structuré car il lui manque l'instruction WHILE. Bien entendu, c'est une affirmation totalement erronée. Il ne suffit pas de faire une boucle "TANT QUE" et de décaler l'intérieur des boucles `for` pour programmer de façon structurée. Et le langage n'a rien à y voir : on peut parfaitement utiliser la programmation structurée sur un programme en assembleur !

Imaginons donc que vous ayiez un tout petit programme à faire, vraiment minuscule. Par exemple, une routine qui ajoute la TVA à un prix quelconque. C'est vraiment simple à réaliser. En Basic, nous obtiendrons un programme comme celui-ci :

```
10 CLS : N=0 : T=0
```

```
20 INPUT "Entrez le prix HT";N
30 T=N*1.186
40 PRINT "Le prix TTC est de : ";T
```

Cela n'en a pas l'air, mais c'est déja de la programmation structurée !

En effet, vous constatez que chaque partie du programme (c'est-à-dire chaque ligne, car c'est un programme simple) a une fonction bien déterminée n'interférant en aucun cas avec les autres :

- la ligne 10 est destinée à initialiser le programme et son environnement : effacement de l'écran et mise à zéro des variables (tout à fait facultative, c'est pour la démonstration);

- la ligne 20 procède à la saisie des données de l'utilisateur.;

- la ligne 30 effectue les calculs;

- la ligne 40 affiche le résultat de l'algorithme (puissant) de la ligne 30.

C'est ce qu'on appelle un programme structuré car il est basé sur une STRUCTURE claire et facile à modifier. Si par exemple demain le taux de TVA retombe à 17.6 %, il n'y aura rien à changer dans la partie saisie des données, ni dans celle de l'affichage.

Par opposition, voici un programme qui fait la même chose mais qui n'est pas du tout conçu structuré. C'est plus le travail d'un débutant possédant son ordinateur depuis trois jours (stade par lequel nous sommes tous passés !) :

```
10 CLS:INPUT "Entrez le prix HT ";N
20 N=N*1.186:PRINT "La prix TTC est de :";N
```

Outre le fait que certaines parties indépendantes sont regroupées ensemble (ce qu'il faut toujours éviter pour s'y retrouver plus facilement), l'utilisation de la variable N dans plusieurs tâches simultanées est une hérésie, génératrice d'effets de bords et d'erreurs probables, même dans un si petit programme.

Expliquons nous.

Certes, le second programme occupe probablement moins d'octets en mémoire, et tourne très légèrement plus vite car il ne possède qu'une seule variable et évite à l'interpréteur Basic deux passages à la ligne. Mais cet argument, valable et souvent imparable il y a six ou sept ans sur des ordinateurs aux maigres 16 Kilos de mémoire, n'est plus tout à fait d'actualité. De toute façon, si l'on utilise un compilateur, le code compilé sera sensiblement le même dans les deux cas.

Il est évident qu'un tel programme ne posera de toute façon aucun problème de relecture, de mise au point, ou de maintenance. Il est si petit ! Mais imaginez par contre un imposant programme de comptabilité ou de jeu, où le programmeur s'amuserait ainsi à mélanger les instructions sur une seule ligne, en utilisant une

dizaine de variables identiques pour des taches différentes. Imaginez encore qu'une erreur survienne. Combien va t-il falloir de temps avant de trouver où se trouve le bug, et quelle variable ne prend pas la bonne valeur ?

Ne tentez pas le calculer, car ce genre de programmation anarchique a pour caractéristique principale la génération d'un autre bug à chaque correction !

La programmation structurée, c'est avant tout deux principes fondamentaux :

1) A chaque tâche son bloc de programme.

2) A chaque bloc de programme ses variables.

Mais ce n'est pas tout.

Car cela, c'est le côté programmationde la chose. Mais la programmation structurée ne commence pas au clavier. Elle entrer en action dès que le problème est posé.

En effet, quelle que soit la méthode utilisée, la réalisation d'un programme se découpe en trois composantes essentielles : CONCEPTION, PROGRAMMATION, MISE AU POINT. La programmation Basic telle qu'on l'apprend par soi-même lorsqu'on débuteentraine vite, sur de gros programmes (entendez par là : trop gros pour cette méthode), le travail suivant :

- un peu de conception;
- programmation directe;
- découverte de quelques bugs;
- un peu plus de programmation;
- découverte d'autres bugs suite aux corrections;
- encore un peu plus de programmation;
- découverte d'autres bugs suite aux corrections;
- de plus en plus de programmation;
- découverte d'autres bugs suite aux corrections;
- ....
- et abandon du projet, ou travail bâclé et programme fourmillant de bugs cachés.

Cette méthode a un charme certain. C'est le plaisir insoupçonné de la chasse au bug, sport des plus passionnants. Mais malheureusement, la méthode "nuit-blanche-café-noir", malgré son indéniable aspect artistique, limite considérablement la complexité des programmes que l'on peut ainsi développer, ainsi que leur évolution. Certains auteurs perdent même le fil au cours de la réalisation et mettent des mois à analyser leur propre programme pour retrouver la façon dont il fonctionnait !

La programmation structurée a pour but de passer plus de tempss ur le papier que sur le clavier. Pour cela il existe des méthodes simples, peut-être difficiles à maitriser sur les gros programmes, mais incontestablement efficaces.

Durant la période de conception d'un programme, la démarche est la suivante :

- Tout d'abord établir une maquette. Cela peut être fait soit à l'aide d'un programme simulant le programme final, soit à l'aide d'un manuel utilisateur et technique qui précise exactement tout ce que fera le programme, et grosso-modo, comment il s'y prendra, une sorte de cahier des charges. Beaucoup de choses seront modifiées lors de la mise au point, mais cette étape permet de cerner précisément où l'on va, et comment l'on s'y prend. Elle permet aussi de déterminer quels sont les moyens nécessaires (choix du langage, du mode graphique, etc).

- Décomposer les problèmes en tâches indépendantes.

- Décomposer chaque tâche en sous-taches indépendantes.

- Décomposer etc.... jusqu'à ce que châque tache soit suffisamment élémentaire. A ce niveau, il n'est pas encore question d'un langage particulier. C'est du bon Français, ou bien à l'extrême limite, du "pseudo-code", c'est à dire un langage à tournure informatique, mais sans plus.

- Parallèlement à cela, préciser et définir les structures de données utilisées, les paramètrès qui entreront, les résultats qui sortiront, les aspects de l'écran, etc... Ce travail est souvent réalisé en premier car il donne une idée précise des problèmes ponctuels, et il permet de cerner le problème. Mais on gagne en facilité si on l'effectue au cours de l'analyse et non avant.

Une fois ceci mené suffisamment loin (il ne faut pas y passer sa vie, tout de même), lorsque l'on estime que le découpage des tâches en sous-tâches et des sous-tâches en sous-sous-tâches (et ainsi de suite) est assez fin pour permettre la programmation, ALORS SEULEMENT on s'installe au clavier.

Là encore, la méthode diffère. En effet, il n'est plus question d'attaquer la programmation de l'ensemble ! Il faut au contraire commencer à programmer les plus basses des sous-sous-...-sous-sous tâches. Ces petites routines qui doivent, par conception même, être élémentaires à programmer, et doivent également passer au travers d'une série de tests systématiques.

Lorsque toutes les sous-tâches d'une tâche sont testées et parfaitement au point, alors seulement on passe au niveau supérieur.

On renouvelle ainsi le processus jusqu'à parvenir au programme principal. Et là, théoriquement, il n'y a plus aucune mise au point, puisque si l'on arrive en haut, par principe, c'est que tout ce qui est en dessous est au point ! C'est d'une logique implacable.

SOYONS CONCRETS

Imaginez un programme de jeu. Dans ce programme, il y aura par exemple une routine pour interpréter l'état du clavier, une autre pour vérifier la fin de la partie, une autre encore pour modifier la position d'un personnage sur lécran en fonction de la touche demandée.

Il est évident qu'avant de programmer la routine d'interprétation du clavier, il faudra déja disposer d'une routine qui lit purement et simplement ce dernier. Et celle-ci devra être au point, sinon comment tester celle qui l'utilisera ?

De même, il serait stupide de programmer la routine s'occupant du déplacement du personnage si nous ne disposons pas d'une routine inférieure qui sait afficher un personnage à un endroit précis de l'écran. Il sera ainsi facile de vérifier celle qui déplace le personnage ! Si cela ne fonctionne pas, nous saurons au moins que cela ne provient pas de la routine d'affichage, mais de celle de déplacement .

Vous voyez, c'est en fait tout à fait naturel.

Cette méthode apparemment sans faille n'est bien sûr pas parfaite.

Bien que le principe soit simple, il s'avère à l'usage que la maîtrise de la programmation structurée exige beaucoup de travail, et surtout une attention soutenue lorsqu'il s'agit de gros programmes. Car si la méthode diminue véritablement les phases de mise au point et réduit pratiquement à néant les bugs "incompréhensibles", il est aussi incontestable qu'elle demande au concepteur de connaitre son analyse sur le bout des doigts et elle multiplie le nombre de variables utilisées. De plus, elle oblige le concepteur à respecter une sorte d'auto-discipline. Il faut rester vigilant car la tendance "Cela, c'est simple, je le programme directement" est elle aussi très naturelle et ruine souvent le reste du travail !

En effet, en programmation structurée, le but du jeu (!) est de ne programmer que des routines indépendantes les unes des autrès, travaillant avec des variables qui leur sont propres (locales) et ne modifiant si possible aucune variable extérieure (globale). Si une routine doit utiliser une variable extérieure,cette dernière sera le plus souvent considérée comme un paramètre, et sera donc explicitement fournie à cette routine pour qu'elle puisse clairement l'utiliser (surtout si elle doit la modifier).

Dans la pratique, il y a toujours des cas où il est fastidieux et peu rentable de passer en paramètrès toutes les variables devant être modifiées ou utilisées. Il y a aussi des cas où la fonction est tellement spécialisée, qu'il est inutile de lui passer des paramètrès : cela encombrerait la pile lors de l'exécution et ralentirait les opérations, sans faciliter ni la mise au point ni la relecture du programme. Dans ce cas, on s'arrange tout de même pour éviter les ambigüités.

Petit à petit, on finit vite par adopter la programmation structurée car, même si elle exige un peu plus de rigueur, il s'avère qu'elle ne gâche absolument pas le plaisir de programmer (pour ceux qui le ressentent), qu'elle fait effectivement

gagner beaucoup de temps, qu'elle facilite les modifications ultérieures d'un programme, et surtout qu'elle permet à un même cerveau de concevoir des choses beaucoup plus solides !

Car l'avantage de cette méthode, qui n'apparaît pas tout à fait au début, est de simplifier l'analyse d'un problème, tout en facilitant sa mise au point et donc sa programmation. Si vous programmez plus facilement les choses, vous irez, avec les mêmes efforts, beaucoup plus loin. De plus, cet effet est "récurrent" : plus vous faites de programmes ambitieux, plus vous êtes à même d'en concevoir d'encore plus ambitieux. C'est tout à fait intéressant !

En fait, il s'agit d'une autodiscipline à adopter. Une fois le pas franchi, les premiers progrès sont foudroyants. Il n'est pas rare de constater qu'on réalise en un mois quelque chose que l'on aurait peut-être mis trois mois à faire auparavant. Ensuite, l'écart diminue car, au fur et à mesure que l'on progresse, on conçoit également des choses beaucoup plus complexes, probablement irréalisables par l'ancienne méthode. Ensuite, c'est l'escalade de la puissance : on conçoit de plus en plus fort, et finalement on se rend compte que l'on atteint les limites du tolérable pour un cerveau humain, alors on programme à deux, puis à trois, puis en équipe de cinq ou six... C'est probablement sans fin , et cela déborde du cadre de cet ouvrage.

Tout ceci reste peut-être assez vague pour vous. Quoi qu'il en soit, vous devez au minimum avoir une idée globale de ce qu'est la programmation structurée. Nous examinerons dans l'annexe C la conception d'un programme assez simple pour que vous cerniez encore mieux le sujet.

DERNIERES REMARQUES

Enfin, un mot ultime mais important. Selon une expression consacrée entendue souvent, le langage Basic ne favorise pas la programmation structurée . C'est une absurdité, car le langage de programmation n'apparaît qu'après l'achèvement quasi-complet de l'analyse.

Ce qui signifie, en clair, que même si d'éminents spécialistes prétendent le contraire, la programmation structurée s'accomode très bien de langages comme Basic, Fred de FrameWork II, ou le langage de Dbase III. L'ultime hérésie, vis à vis de ces spécialistes, est d'affirmer avec force qu'on peut parfaitement programmer en assembleur structuré. C'est même en fait la démarche la plus réaliste !

La programmation structurée est une méthode de travail, pas une méthode d'utilisation du C et du Pascal ! Elle concerne l'analyse et le découpage du problème, et non le langage utilisé pour sa programmation.

Et nous terminerons ce bref exposé sur la programmation structurée par une remarque : le passage de l'analyse à la programmation se fait effectivement plus ou moins automatiquement suivant le langage (mais pas plus difficilement). Avec le langage C, point de soucis : c'est pratiquement instantané. Et donc autant l'utiliser !

CHAPITRE 11

FONCTIONS ET DECLARATIONS

Un programme en C est constitué d'un ensemble de fonctions, dont l'une se nomme `main()`. Mais le langage admet une structuration plus poussée. Nous avons constaté l'existence de deux sortes de variables : les variables locales, et les variables globales.

La structuration d'un programme peut se fonder à la fois sur une organisation des variables ou des données, et sur celle des fonctions.

Avant toute chose, nous allons parler des fonctions.

UN EXEMPLE DE FONCTION

Si l'on pousse le raisonnement de la structuration assez loin, on constate vite que le découpage des tâches en fonctions, malgré ses qualités, conduit à un foisonnement d'identificateurs et de fonctions. Si l'on n'y prend garde, on se retrouve finalement avec un programme plus difficile à suivre que s'il était programmé d'un bloc !

Le travail de structuration ne consiste donc pas uniquement à isoler les fonctions sur plusieurs niveaux. Il faut également soigneusement travailler l'utilisation de celles-ci.

D'un point de vue formel, nous pouvons définir la fonction idéale comme une "boîte noire": on lui fournit certaines données en entrée et elle renvoie certains résultats en sortie. Le travail du bon programmeur est de remplir cette boîte avec le mécanisme approprié, de façon à transformer les entrées en résultats sans avoir ultérieurement à se préoccuper du mécanisme en question ni de son fonctionnement.

Ainsi, lorsque vous utilisez dans un programme la fonction `printf`, vous n'avez pas réellement envie de savoir comment elle fonctionne, et surtout vous ne devez pas en avoir besoin.

Une fonction est donc une boîte noire. C'est une bonne chose. Mais quels outils C met-il à notre disposition pour utiliser cette boîte ?

La première des facilités, que nous avons déjà plusieurs fois utilisée, est le passage de paramètres.

Soit la fonction suivante (volontairement programmée de façon peu astucieuse):

```
double carre(double x)
{
  double resultat;
  resultat = x * x;
  return( resultat );
}
```

Nous constatons qu'elle se décompose de la façon suivante :

```
déclaration de la fonction avec son nom (identificateur)
{
  déclaration des variables locales
  instructions
  renvoi d'un résultat
}
```

Dans un programme, on peut grâce à ces éléments utiliser directement la fonction carre dans des expressions, en lui fournissant soit une expression soit une constante, soit une variable, etc. Le travail effectué est simple : nous récupérons, par l'appelation `carre(y)`, le carré de y. Nous n'aurons pas à nous soucier des variables utilisées par la fonction pour effectuer ce calcul, puisqu'elles sont locales (ici, la variable `resultat`). Enfin, la déclaration permet de connaître rapidement, dans un listing, les particularités de la fonction `carre` : elle est de type double (c'est-à-dire quelle renvoie un résultat de ce type), et accepte un paramètre de même type.

ASPECT GENERAL D'UNE FONCTION

Une fonction, qu'il s'agisse de `main()`, ou d'une fonction particulière, répond toujours à une structure bien définie. Nous venons d'en voir un exemple .

1. déclaration de la fonction
2. {
3. déclaration de variables locales
4. instructions
5. renvoi du résultat
6. }

En réalité, les lignes 3, 4 et 5 sont optionnelles : une fonction peut très bien ne rien faire du tout !

ANSI, KERNIGHAN/RITCHIE ET TURBO C

La ligne 1 constitue un résumé des particularités de la fonction, à l'usage du compilateur. Elle admet deux sortes de syntaxes, nommés "classique" et "moderne" par le manuel Borland ! En fait, il existe deux façons de déclarer les fonctions : la manière de Kernighan & Ritchie, telle qu'elle fut définie par les inventeurs du langage, et la façon ANSI, telle que l'a définie le comité chargé de cette normalisation.

Bien évidemment, la déclaration classique est plus répandue, mais la norme ANSI apporte de gros avantages. Nous conseillons (tout comme Borland) d'oublier la déclaration standard et d'utiliser systématiquement la norme ANSI, car elle rend les choses beaucoup plus propres, plus claires; de plus elle facilite le travail de Turbo-C, qui est alors à même d'indiquer les erreurs possibles et d'interdire ce qui ne serait pas correct. La méthode K&R ne permet pas de détecter les erreurs à ce niveau.

Fonction type K&R :

```
[ classe ] [ type ] identificateur()
type paramètre1;
type paramètre2;
...
{
  bloc;
}
```

Fonction type ANSI et BORLAND

```
[ classe ] [ type ] identificateur( type paramètre, ....)
{
  bloc;
}
```

La grosse différence est la réunion sur une seule ligne des paramètres qu'accepte la fonction, et surtout leur insertion dans les parenthèses de cette ligne. Cette déclaration est donc beucoup plus lisible, il faut bien l'avouer !

Lors des compilations, avec les déclarations de type ANSI, il est ainsi possible de vérifier de façon complète l'utilisation des fonctions au sein du programme : type et nombre de paramètres passés, etc...

Notez que Turbo C reconnaît automatiquement les deux types de déclarations. Mais dans le cas K&R, il ne donnera aucun avertissement , pour vous mettre en garde contre une possibilité d'erreur. Dans certains cas, il faudra attendre l'exécution du programme ou le Link pour détecter une erreur de passage de paramètres. Si le programme est long, que de temps perdu ...

La déclaration est constituée, dans les deux cas, d'une partie concernant la classe et le type de la fonction, et d'une autre destinée à la définition des paramètres que reçoit la fonction.

CLASSE ET TYPE DE LA FONCTION

`classe type identificateur` définit la fonction, sa classe, et le type du résultat qu'elle renvoie. Les classes possibles sont `auto` et `static`. Ignorons allègrement la classe "static", son utilité est strictement réduite à des considérations d'encombrement mémoire qui ne nous concernent pas vraiment ici.

Par défaut, une fonction est accessible par tout le code qui suit sa définition (ou sa déclaration de prototype, nous verrons cela dans la suite de ce chapitre). C'est la classe `auto`, ou automatique.

Il est donc inutile de préciser la classe de la fonction, à moins que vous ne vouliez vraiment utiliser une fonction `static`. Dans ce cas, reportez-vous au manuel Borland pour en savoir plus.

En revanche, le type de la fonction est important. Si vous omettez ce type, il sera implicitement assimilé à `int`, comme pour les variables. Mais vous pouvez (et nous le conseillons) également préciser tout type reconnu par C. Nous reviendrons sur les types dans le chapitre sur les données. Mais vous en connaissez déjà quelques uns : `int`, `float` et `char`. Ces types sont parfaitement utilisables pour le résultat d'une fonction.

LE TYPE void

Il existe d'autre part un nouveau type de fonction utilisable, astucieusement inventé par ANSI : le type `void`, ou "rien-du-tout". Il permet de programmer une fonction ... qui ne renvoie aucun résultat ! En un sens, une fonction `void` est semblable à une procédure du Pascal.

Quel intérêt, direz-vous ? Eh bien par exemple, une fonction qui efface l'écran n'a aucun résultat à renvoyer. Dans ce cas, on peut la déclarer avec le type "void", ce qui évitera lors de son appel une réservation de mémoire en pile et de nombreux tests, et rendra le code obtenu plus compact. Cela permettra aussi d'interdire une utilisation inattendue de "return" au sein de cette fonction, ou son appel dans une expression. Cela facilite encore une fois le travail de compilation et de correction des erreurs ou des étourderies.

LES PARAMETRES

La partie suivante de la déclaration de fonction indique au compilateur quelles données vont être envoyées à la fonction lors de son appel, et quel sera le type de ces données. Elle permet également d'associer un identificateur à chacune de ces données, de façon à les référencer simplement dans le corps de la fonction.

Reprenons notre fonction de mise au carre :

DECLARATION ANSI :

```
double carre ( double x )
```

DECLARATION K&R :

```
double carre()
double x;
```

Dans cette déclaration, nous constatons que la fonction recevra un seul paramètre, de type `double`, et que ce paramètre sera appelé `x` dans les instructions de la fonction.

Lors de l'appel d'une fonction, C calcule d'abord la valeur de l'expression passée en paramètre, puis il associe cette valeur à l'identificateur du paramètre dans la fonction.

Ne confondez pas les paramètres et les variables locales ou globales!

Cette distinction est très importante. Dans la suite de la fonction carré, l'identificateur x représente la valeur passée en paramètre ou le résultat d'une expression comme dans :

```
y = carre (toto * pi * 15.0 /100);
```

Mais en aucun cas `x` n'est assimilable à une variable. En effet, vous pouvez utiliser `x` dans une expression, dans des tests, mais vous ne pouvez absolument pas le modifier : un paramètre est une valeur. Il représente une constante, un nombre, une chaîne, ou tout autre valeur, mais n'est pas une variable.

Cette notion est très importante : cela signifie entre autres que vous ne pouvez pas modifier le contenu d'une variable simplement en la passant en paramètre à une fonction. C'est là une particularité du C, qui le distingue par exemple du Pascal. Dans ce dernier, en effet, on peut passer un paramètre par valeur o par variable; dans ce dernier cas, la fonction a le droit de modifier le paramètre qui lui est passé. Mais il est fréquent d'assister alors à des effets imprévus : si une fonction tente de modifier un paramètre alors que celui-ci n'a pas été passé par variable, il restera en fait inchangé; ce type d'erreur est très difficile à détecter. En C, ce problème n'existe pas, car il est impossible de passer une variable en paramètre. On ne peut envoyer que son contenu.

Cette apparente restriction est en fait un acte de bienveillance de la part du langage. Elle évite de grosses erreurs d'étourderie.

Bien entendu, si l'on désire réellement modifier un paramètre à l'intérieur d'une fonction, il suffit d'utiliser des pointeurs pour remédier au problème; mais cela oblige justement le programmeur à être pleinement conscient de ce qu'il tente de faire ! En voici un exemple.

Modification d'une variable par une fonction

Supposons que nous voulions remodeler la fonction `carre` : au lieu de renvoyer le carré d'un nombre, nous voulons lui envoyer une variable, et la fonction devra mettre cette variable au carré.

La première idée est d'écrire ceci :

```
/*----------------------------------------*/
/* DECLA.C                    * /
/* Exemple de passage de paramètre incorrect */
/*----------------------------------------*/


carre( short x)
{
  x = x * x;
}

main()
{
  short test = 2;
  carre(test);
  printf("%d\n",test);
}
```

Malheureusement, cela se compile sans problème, mais ne modifie absolument pas la variable `test`.

En effet, comme nous l'avons expliqué, dans la fonction carre, `x` représente non pas la variable `test`, mais la valeur que cette dernière contient, c'est à dire "2". Vous constatez d'ailleurs que l'instruction `x = x * x` est théoriquement un non sens, puisqu'elle procède à une assignation `2 = 2 * 2` ! (En réalité, la valeur 2 est copiée dans une sorte de variable locale de nom `x`, mais autant oublier ce détail car le principe est le même).

Il faut donc procéder autrement. D'autre part nous allons rendre notre déclaration plus efficace (vous aviez remarqué la déclaration selon K&R? bravo).

Voici la bonne version de ce programme :

```
/*-------------------------------------*/
/* DECLA2.C                            */
/* Passage de paramètre par pointeur   */
/*-------------------------------------*/

void carre( short *x)

{
  *x = *x * *x;
}
```

```
main()
{
  short test = 2;
  carre(&test);
  printf("%d\n",test);
}
```

Notez dans la déclaration de la fonction `carre` le type `void` : la fonction de renvoie aucun résultat, elle va modifier une variable extérieure. L'exécution de ce programme affiche bien "4" et non plus "2" : la variable `test` a donc bien été mise au carré.

Dans la fonction carré, le paramètre déclaré est `short *x`, indiquant que `x` représentera non pas une valeur `short`, mais un pointeur vers une valeur de ce type.

Dans le corps de la fonction, on peut alors modifier et utiliser la valeur de cet entier en utilisant l'indirection `*x`. Mais remarquez que cela modifie aussi l'appel de la fonction depuis `main()`, car il faudra à présent lui passer `&test` en paramètre. En effet, il ne faut pas oublier désormais que la fonction `carre` prend une ADRESSE en paramètre (un pointeur), et non plus le résultat d'une expression. Bien sûr, cela interdit maintenant un appel direct comme "`carre (expression)`". C'est une contrainte nécessaire : il est en effet difficile d'imaginer comment mettre une expression au carré ! Il faudra donc stocker le résultat de l'expression dans une variable avant d'appeler `carre` avec le pointeur de cette variable.

Passer un tableau en paramètre

Il est inutile d'utiliser les opérateurs `&` et `*` pour passer un tableau en paramètre, puisque l'identificateur d'un tableau représente implicitement l'adresse de celui-ci. Souvenez-vous que cela est également vrai avec les chaînes de caractères. Il faut donc retenir que, à la différence des variables simples, on peut toujours modifier un tableau ou une chaîne de caractères passés en paramètre à une fonction.

PROTOTYPES

Pour clore cet exposé sur les déclarations, nous allons examiner une facilité offerte par la norme ANSI et Turbo C : les prototypes de fonctions.

Un prototype est en fait tout simplement une copie de la déclaration de la fonction en début de programme, à une différence près : un point-virgule la termine, contrairement à la véritable déclaration.

Le prototype a un rôle bien précis : faciliter la détection d'erreurs possibles par le compilateur, et rien de plus. Le programme sera exactement le même, excepté à l'étape de la compilation. Le prototype se place en début de programme. Il y en aura un pour chaque fonction déclarée ensuite.

Le programme prend donc l'allure suivante :

```
/*prototypes*/
type fonction1 (type paramètre, ....) ;
type fonction2 (type paramètre, ....) ;
...

/*déclarations des fonctions*/
type fonction1 (type paramètre, ...)
{
  corps de la fonction1
}
type fonction2 (type paramètre, ...)
{
  corps de la fonction2
}

/*programme principal*/
main()
{

  ...
}
```

Comme la majorité des normalisations de l'ANSI, la mise en place de prototypes est optionnelle. Ce n'est absolument pas une obligation, mais utilisez-les systématiquement. En effet, leur utilisation n'a que des avantages :

- Ils sont regroupés tout en début de programme, et peuvent donc être supprimés facilement si l'on désire utiliser un compilateur non-ANSI.

- A partir du moment où un prototype est présent, la fonction peut être déclarée à la façon "K&R" avec les mêmes avantages qu'à la façon "ANSI". On peut donc avoir un programme totalement compatible avec un autre compilateur non-ANSI une fois les prototypes écartés du source.

- Ils ne modifient en rien le code généré.

- Ils constituent une en-tête de listing extrêmement intéressante, résumant les particularités importantes de chaque fonction dans le programme sans que l'on soit obligé de chercher ces fonctions dans le reste du listing.

- Ils permettent à Turbo C de détecter les erreurs d'appel ou les possibilités d'erreurs au passage des paramètres.

- A partir du moment où le prototype d'une fonction a été déclaré, et même si la fonction se trouve en fin de programme, les autres fonctions peuvent l'utiliser. C'est la seule façon d'obtenir une telle souplesse. La mise en place des prototypes permet de ne plus se préoccuper de l'ordre des déclarations et définitions de fonctions.

Exemple : supposons que nous disposions d'une fonction `calcul`, qui utilise une autre fonction `carre`. En C standard, nous sommes obligés d'écrire ces fonctions dans l'ordre suivant :

```
fonction carre
  ...
fonction calcul
...
fonction main
...
```

Avec des prototypes, cet ordre n'est absolument plus obligatoire. L'ordre des prototypes lui-même est sans importance également. Par exemple nous pouvons procéder comme ceci :

```
prototype de la fonction calcul
prototype de la fonction carre
...
fonction main
...
fonction calcul
...
fonction carre
...
```

Illustration

L'utilisation d'un prototype ne présente guère d'intérêt dans notre programme carre, mais voici tout de même le listing modifié :

```
/*-----------------------------------------*/
/* PROTO.C                                 */
/* Exemple de prototype de fonction        */
/*-----------------------------------------*/


/* prototype de la fonction carré (notez le ";" final) */

void carre( short *x);

main()
{
  short test = 2;
  carre(test);
  printf("%d\n",test);
}

void carre(short *x)
{
  *x = *x * *x;
}
```

Si vous modifiez quoi que ce soit dans la déclaration de la fonction carre par rapport au prototype, vous obtiendrez une erreur du compilateur. C'est bien pratique lorsque la fonction se trouve hors de vue de l'erreur : dans ce cas il suffit de regarder en début de listing pour vérifier si l'appel est correct. Par exemple, si nous supprimions ci-dessus le `&` dans l'appel de la fonction `carre`, nous obtiendrions le message erreur suivant :

```
Error proto.c 8: Type mismatch in parameter 'x' in call to
'carre' in function main
```

qui indique un mauvais type pour le paramètre passé à `carre` dans la fonction `main` (il faut passer une adresse ou un pointeur, et ayant supprimé le `&`, nous envoyons une valeur `short`). Ce genre de détection d'erreur est impossible en C standard. Et si l'on en juge d'après le nombre des programmeurs étourdis, c'est une facilité qui est plus qu'appréciable.

En résumé, utilisez les prototypes, ils vous aideront !

MODULES ET LIBRAIRIES

Il arrive qu'un programme soit suffisamment gros pour comporter des parties complètement indépendantes. Par exemple, si un programme comporte une vingtaine de fonctions d'affichage, il peut être judicieux de séparer ces fonctions du programme, de les placer dans un fichier à part. D'un côté, cela éclaircira quelque peu le listing principal, et de plus cela permettra d'utiliser facilement ces fonctions dans d'autres programmes.

Dans ce cas, on recourt un module. Un module est un fichier source C qui ne comporte pas forcément de fonction "main", mais qui ne contient en tout cas que des fonctions de type "boîte noire", c'est à dire des fonctions dont le travail ne dépend que des paramètres qui lui sont passés, à l'exclusion de toute variable extérieure.

Ces fonctions peuvent alors être placées ailleurs, dans un module compilé séparément une fois pour toute. On doit procéder aux opérations suivantes :

Première possibilité :

1. Appeler le fichier source des fonctions ainsi programmées avec un suffixe .H, comme par exemple "routines.H", et placer ce fichier dans le répertoire contenant les fichiers .H de Turbo C (C:\C\INCL ou \INCL, selon notre modèle de configuration).

2. Au début des programmes utilisant ces fonctions, placer une directive:

```
#include <routines.H>
```

3. La suite des opérations est identique (compilation,...).

Remarque : Les fonctions sont recompilées à chaque fois que l'on recompile le programme, et entrent dans le code objet et exécutable de celui-ci. Par contre, elles n'encombrent plus le fichier source, . Mais si le fichier ROUTINES.H comporte par exemple 20 fonctions, elles seront toutes intégrées au code exécutable, même si le programme n'utilise qu'une d'entre elles.

Seconde possibilité :

1. Compiler uniquement le fichier des fonctions (Alt-F9 ou Alt-C et C).

2. Placer le fichier ROUTINES.OBJ obtenu dans le répertoire des librairies Turbo C, soit C:\C\LIB ou \LIB selon notre modèle de configuration.

3. Créer un fichier avec uniquement les déclarations des fonctions, celles-ci étant précédées de `extern` et terminées par un "`;`", afin de créer les prototypes, et l'enregistrer sous ROUTINES.H.

4. Créer, pour un programme utilisant les fonctions, un petit fichier texte de nom "nom.PRJ" et contenant les lignes suivantes:

```
nom du fichier contenant le source du programme

nom du fichier contenant les fonctions compilées
```

5. Utiliser Alt-P (menu "Project"), puis P ("Project Name") pour indiquer le nom de ce fichier PRJ.

6. Inclure une directive au début du source du programme :

```
#include <routines.H>
```

7. Pour une modification du programme, utiliser F9 (make) pour recompiler.

8. Remarques : Cette méthode inclura le code compilé des fonctions dans le programme pour créer l'exécutable. La recompilation du programme n'entraîne plus celle des fonctions. Si une erreur de compilation du programme intervient, le fichier contenant les fonctions n'est même pas lu; il n'entre en action qu'au Link, lorsque la compilation est terminée. D'autre part, seules les fonctions réellement utilisées dans le programme sont incluses dans le code exécutable, même si le fichier ROUTINES.OBJ en comporte d'autres.

Bien que la seconde méthode paraisse moins pratique, elle apporte une rapidité de compilation très appréciable sur les gros programmes, puisque les fonctions déja compilées ne sont que linkées. On en prend vite l'habitude. Plus le programme comporte de fonctions déjà compilées par ailleurs, plus son temps de compilation est faible, donc sa mise au point est plus rapide.

Avantages des modules.

Tout d'abord, ils vous permettent de créer ce que l'on nomme une librairie. Vous pouvez par exemple créer vos fonctions concernant l'affichage, les réunir en un seul fichier, et les isoler une fois compilées (sans les linker). Les fonctions ainsi isolées pourront être utilisées dans n'importe quel programme selon les règles que vous aurez définies. Vous saurez ainsi que ces fonctions fonctionnent correctement, et il n'y aura plus besoin de les recompiler systématiquement. De plus, le programme final (avec la seconde méthode) ne sera encombré que par le code des fonctions réellement utilisées.

Autre avantage corollaire de celui-ci, puisque vous isolez vos fonctions compilées, cela suppose qu'elles sont parfaitement débugguées et au point. Cela facilite donc une grande partie du travail. Et plus vous disposerez de fonctions et de librairies, plus la partie "prête à l'emploi" de vos futurs programmes sera importante, plus vous gagnerez de temps dans la réalisation de ceux-ci.

De même, dans l'idéal, vous disposerez un jour de toutes les fonctions interface-utilisateur et générales imaginables, ce qui vous permet de ne travailler que sur le problème particulier que vous avez à résoudre. Pour une même tâche, vous avez donc beaucoup moins de travail. Vous pourrez donc vous attaquer à des problèmes de plus en plus complexes.

La clarté et l'homogénéité de vos programmes s'en ressentiront également : si par exemple vous avez mis au point une librairie de fonctions pour gérer des fenêtres et des menus déroulants, tous vos programmes pourront les utiliser; les utilisateurs s'y retrouveront donc plus facilement d'un programme à un autre. Et vous bénéficierez aussi, au niveau programmation, d'une connaissance de plus en plus grande de vos fonctions, ce qui amènera vite une mise en œuvre rapide et efficace de toutes ces fonctions.

Enfin, ultime avantage plus subtil mais appréciable dans l'absolu, la mise au point de librairies implique que chaque fonction soit indépendante du programme qui l'utilisera. En conséquence, les fonctions ne doivent utiliser aucune variable externe ou globale (certaines variables globales peuvent cependant être utilisées au sein du fichier contenant la librairie, mais uniquement dans le cas où elles sont vraiment communes aux fonctions, par exemple s'il s'agit d'un environnement sur lequel travaillent toutes les fonctions et que cet environnement est local et n'intervient pas dans le programme utilisateur).

Pour que les fonctions soient indépendantes, il est obligatore de les programmer proprement, et surtout en adoptant la méthode de la

programmation structurée, afin que les inter-dépendances éventuelles de chacune d'entre-elles soient parfaitement définies. De même, vous devrez les avoir parfaitement mises au point avant de les considérer comme utilisable.

La mise au point d'une librairie est donc finalement une chose assez contraignante au premier abord, mais elle est extrêmement rentable à plus ou moins long terme : vous apprendrez à programmer proprement grâce à elles, et vous bénéficierez pour vos programmes d'une facilité de dévellopement très agréable.

CHAPITRE 12

DONNEES ET CALCULS

TYPES SIMPLES

Nous avons vu dans le chapitre Premiers Pas les types `char`, `float`, `int`, ainsi que les tableaux qui les accompagnent.

Dire que nous avons tout vu serait un euphémisme savoureux. Comme nous vous l'avions signalé, ces notions vous ont permis de commencer à programmer, mais elles constituent une infime parcelle des possibilités de C en matière de types de données.

Nous allons maintenant examiner d'un peu plus près les types de données simples reconnus par C.

Tout d'abord, nous devons tout de même vous mettre en garde contre une tentative d'utilisation de tous ces types : certains sont franchement déconseillés (pour cause de non portabilité), d'autres ne présentent qu'un intérêt anecdotique. A quelques exceptions près, les trois types de bases que nous avons examinés (`char`, `int` et `float`) suffisent largement pour la majorité des applications. Mais il peut être intéressant, dans tous les cas, de connaître leur existence et de leur trouver une utilisation valable.

C reconnait essentiellement quatre types de données de base :

- le type `char`, ou caractère;
- le type `int`, ou entier;
- le type `float`, ou réel;
- le type `double`, ou réel double précision.

Nous avons trés brièvement évoqué ce dernier auparavant, lorsque nous nous sommes penchés sur le type `float`. Le type `double` est totalement équivalent à `float`, mais il comporte beaucoup plus de chiffres significatifs (et donc réclame également plus de place en mémoire, car personne n'est parfait).

L'avantage de C, c'est qu'il effectue les calculs réels systématiquement en double précision, donc en `double`. Ceci est vrai sur tous les compilateurs existants, et Turbo C ne fait pas exception à la règle. Lorsqu'un calcul de type `float` est demandé, ses arguments sont transformés temporairement en `double`, puis traités, et le résultat est de nouveau converti, mais en sens inverse, vers le format `float`.

Mais c'est aussi un gros inconvénient : ceci entraine de facto une relative lenteur des calculs, par rapport à l'hypothèse de calculs en réels simple précision.

Voici donc une règle imporatnte : dans la plupart des cas, sachant qu'on n'a pas trop le choix, il vaut mieux utiliser le format `double` pour les nombres réels, car le traitement, paradoxalement, est plus rapide (il ne procède pas aux conversions). De plus, le format `double` est directement compatible, au niveau du stockage, avec le co-processeur 8087/287/387 (du moins avec Turbo C) ce qui accélère les traitements lorsque l'on possède ce processeur et que le compilateur est positionné pour utiliser cette option.

Rappelez-vous que vous ne pouvez obtenir que les calculs se fassent réellement en format `float`. Quoi que vous fassiez, même si vous placez des `cast` partout et que toutes vos variables sont de type `float`, tout ce que vous pourrez obtenir est une diminution significative de l'encombrement mémoire, mais certainement pas une accélération des calculs (au contraire).

LES MODIFICATEURS

Devant un de ces quatre types simples, on peut utiliser un modificateur. Un modificateur est un élément du langage qui, comme son nom l'indique, modifie les caractéristiques du type de base sans en changer la signification.

Les modificateurs sont au nombre de trois :

- `short` et `long` pour le type `int`;

- `long` pour le type `float`;

On dispose en plus de `unsigned` qui est un "sur- modificateur" puisqu'il peut s'appliquer à `short`, `long`, `int` et `char`.

short

Ce modificateur indique d'utiliser un nombre réduit d'octets (2) pour les entiers, ce qui leur autorise les valeurs -32768 à 32768. Sur un IBM et en Turbo C, il est strictement équivalent à `int` seul, car ce dernier utilise également deux octets. Mais sur certains gros ordinateurs ou minis, le type `int` correspond au format entier du processeur, et peut donc utiliser un nombre variable d'octets. `short` assure que nous utiliserons exactement 2 octets pour les entiers. `short` seul est totalement équivalent à `short int` : le `int` est implicite.

En conséquence, dans la suite de cet ouvrage, nous n'utiliserons plus jamais `int`, mais systématiquement `short`. Cela ne change pas grand chose mais autorise une portabilité assurée, et c'est une bonne habitude à prendre.

long

Celui-ci indique d'utiliser soit des entiers sur 4 octets (s'il est seul ou suivi de `int`), soit des réels double précision. Notez les équivalences suivantes :

```
long int   = long

long float = double
```

En conséquence, on retiendra essentiellement que `long` seul permet d'utiliser des entiers dont les valeurs vont de -2 147 483 648 à +2 147 483 647, ce qui est ... beaucoup ! Les autres utilisations de `long` se ramènent soit à `long`, soit à `double` pour les réels. Il existe également `long double`, mais ce type est équivalent à `double` (ne rêvons pas !). Les constantes (entières ou réelles) de type long peuvent être suffixées avec un "L" majuscule pour forcer ce format; par exemple :

```
float   fpi = 3.14159;

double  dpi = 3.141592659L;
```

unsigned

Dans les types vus jusqu'à présent, un des bits est réservé au signe. Par exemple, pour le type `short`, il y a 15 bits pour stocker le nombre et 1 pour son signe. Le modificateur `unsigned` permet d'affecter ce bit au nombre, ce qui double la valeur maximale stockable, mais interdit les nombres négatifs, et modifie quelque peu les résultats de certaines opérations.

Ce modificateur est essentiellement utilisé avec le type `char` pour accéder aux codes 128 à 255 sans avoir à préciser -128 à -1, et avec les paramètres de routines en langage machine pour récupérer le contenu de registres 8 ou 16 bits sous une forme standard. Il existe :

`unsigned char` : les caractères ont les codes 0 à 255 (un octet)

`unsigned int` : 16 bits sur l'IBM, donc 0 à 65535 (processeur 16 bits)

`unsigned long` : 32 bits, donc 0 à 4294967295 (4 octets)

`unsigned short` : 16 bits, 0 à 65535.

Corollaire : le type `unsigned` permet d'utiliser des variables entières positives ou binaire sur 8, 16 ou 32 bits (respectivement `unsigned char`, `unsigned short`, `unsigned long`). Une constante de type `unsigned` peut être suffixée avec un "U" majuscule, comme par exemple :

```
unsigned short gros_max = 15U;
```

Maintenant que nous connaissons les modificateurs, voici un résumé des types simples et de leurs caractéristiques. Rappelons que le type `char` peut stocker des caractères (comme dans `char c='R';`) ou des valeurs numériques 8 bits (comme dans `char c=97;`).

```
          char   : caractère. 1 octet, 8 bits, -128 à +127.
unsigned char    : caractère. 1 octet, 8 bits, 0 à +255.

          int    : Suivant machine (IBM-PC:2 octets) cf short.
unsigned int     : Suivant machine (IBM-PC:2 octets) cf unsigned short.

          short  : entier.   2 octets, 16 bits, -32767 à 32768.
unsigned short   : entier.   2 octets, 16 bits, 0 à +65535.

          long   : entier.   4 octets, 32 bits, -2147483648 à 2147483649.
unsigned long    : entier.   4 octets, 32 bits, 0 à 4294967295.

         float   : réel.    4 octets, 32 bits, 3.4e-38 à -3.4E+38
         double  : réel.    8 octets, 64 bits, 1.7e-308 à -1.7e+308
```

Dès à présent, vous remarquez que le type `int` est clair, mais il dépend de la machine. Cependant, beaucoup de librairies utilisent ce format. Nous conseillons plutot d'utiliser `short` si vous désirez des programmes transportables, car ce type (identique à `int` avec Turbo C) possède dans tous les cas 16 bits.

D'autre part nous n'avons pas inclus dans ce tableau les déclarations inutiles ni implicites comme `long float` ou `short int`. Elles sont expliquées plus haut dans les paragraphes des modificateurs.

Avec tous ces types, nous pouvons déclarer ou utiliser des variables et des tableaux comme vu précédemment. Ainsi, vous pouvez par exemple entrer, compiler/linker et exécuter le programme suivant :

```
main()
{
  double    d[2];
  float     f[2];
  f[0] =0.0000000000001;
  d[0] =0.0000000000001L;f[1] = 1.0/f[0];
  d[1] = 1.0L/d[0];
  printf("%14.2f float\n%14.2f double\n",f[1],d[1]);
}
```

L'exécution vous permet de mieux saisir la nuance entre les types `float` et `double`. Remarquez les chaînes de formatage dans `printf` (`14.2` signifie 14 chiffres avant la virgule, et deux après).

Les déclarations des variables avec ces nouveaux types suivent exactement les mêmes règles que celles les autres. Par exemple, voici quelques définitions correctes :

```
unsigned char   uc1_test = '\t';
unsigned char   uc2_test = 240;
unsigned int    ui_test  = 60000;
long             l_test  = -1000000;
double           d_test  = 234.23598759873259813024986543L;
float            f_test  = 234.23654276187264982106498210;
```

CLASSES DE VARIABLES

Pour des raisons de structuration, nous avons déjà suggéré la notion de localité de certaines variables. En effet, en langage C comme en Pascal, les variables peuvent, suivant leur emplacement dans un programme, être plus ou moins visibles au sein de celui-ci.

La notion de visibilité des variables que nous allons maintenant examiner est extrêmement importante dans tout langage de type structuré. Elle facilite grandement le travail du programmeur, en lui permettant de prévoir les éventuels effets de bord et d'y remédier par avance !

Schématiquement, disons qu'on peut, avec C, rendre n'importe quelle variable visible uniquement du bloc qui la contient. C'est même trés simple.

Rappelons d'autre part qu'un programme peut comporter plusieurs fichiers indépendants, compilés et linkés ensemble. De tels fichiers sont appelés modules, et un module est totalement assimilé à un bloc, tout comme le bloc d'une fonction ou un bloc d'instruction. Cette notion est importante car il peut être utile de comprendre qu'une variable globale, donc déclarée à l'extérieur de toutes les fonctions qui s'y trouvent, est également locale, mais locale à ce module, donc inaccessible en temps normal aux autres modules. Nous verrons comment outrepasser cette localité, et pourquoi ne pas le faire !

Enfin, une variable, ainsi qu'une fonction, n'existent qu'à partir du moment où elles sont déclarées : ceci explique pourquoi une variable globale doit être placée avant les blocs qui l'utilisent. Ainsi, est-il aussi possible de définir des variables globales visibles uniquement à l'endroit où elles commencent à être utiles, de façon à prévoir encore mieux les effets de bord. Nous y reviendrons également.

Examinons le programme suivant, où les numéros de ligne ne sont là que pour faciliter les explications qui suivent :

```
1   int i=1;
2   main()
3   {
4         int i=2;
5         int j;
6         for (j=1; j<=1; j++)
7         {
8         int i=3;
9         }
10 }
```

Dans ce programme totalement inutile (il ne fait rien !), nous possédons non pas une mais bien trois variables nommées `i`.

La première est déclarée en ligne 1 : il s'agit d'une variable globale à tout le programme. Nous y reviendrons.

La deuxième variable, déclarée en ligne 4, n'écrase pas la première, mais la rend invisible tant que le bloc de visibilité de cette nouvelle variable `i` est en cours. C'est à dire que la première variable `i` redevient disponible (avec la valeur 1) à partir de la ligne 10. Si le programme continue avec une autre fonction, nous pouvons l'y utiliser. Même principe pour la seconde variable, qui est "cachée" par celle de la ligne 8, et qui est donc invisible des lignes 7 à 9.

Si nous pouvions afficher la valeur de la variable `i` visible à chacune de ces lignes, nous obtiendrions ceci :

N° ligne de programme	Valeur de `i` visible :
`1 int i=1;`	1
`2 main()`	1
`3 {`	1
`4 int i=2;`	2
`5 int j;`	2
`6 for (j=1; j<=1; j++)`	2
`7 {`	2
`8 int i=3;`	3
`9 }`	2
`10}`	1

Mais ce n'est pas aussi élémentaire que cela. En effet, en plus de ces possibilités, C autorise plusieurs autres types de visibilité et de stockage au sein de la mémoire, que l'on regroupe sous l'appellation de classe de stockage , ou tout simplement classe.

Les différentes classes sont associées à un nom, lequel se place dans la déclaration des variables juste avant le type (comme les modificateurs, mais avant ceux-ci s'il y en a). Nous avons donc, généralement, une déclaration de variable qui possède la structure suivante :

```
classe modificateur type nom_de_variable ;
```

Les différentes classes disponibles sont examinées ci-après.

auto

Il s'agit de la classe implicite. La zone de stockage d'une variable `auto` (pour automatique) est à l'intérieur du bloc qui la contient, et n'est créée que lors de l'exécution. Ce bloc peut être aussi bien un bloc d'instruction que le bloc d'une fonction. Reportez-vous à l'exemple ci-dessus : les variables `i` contenant 2 et 3 sont de classe `auto`. La variable `i` contenant 1 n'est située dans aucun bloc de fonction ou d'instruction, mais elle est tout de même de type `auto`. Elle est cependant créée non pas à la compilation (voir la classe `static` ci-dessous) mais lors de l'exécution. Notez également qu'à la sortie du bloc, une variable automatique est détruite et n'est plus disponible.

static

Cette classe est destinée à créer des variables qui resteront en place durant toute l'exécution, car elles sont intégrées au code lors de la compilation, avec leur emplacement réservé. Ces variables ne peuvent pas être perdues, et elles sont par défaut (sauf initialisation particulière) initialisées à zéro par le compilateur.

Une variable statique est accessible dans tout le programme qui le suit, même si elle est déclarée à l'intérieur d'un bloc d'instructions (ce que l'on évitera). Exemple :

```
main()
{
  static short a=1;
  short b=2;
}
...
```

Dans les autres fonctions du programme, la variable statique `a` existe toujours et contient la valeur 1, tandis que la variable `auto b` disparaît dès que `main()` est achevée. Mais attention, `a` n'est "visible" qu'à l'intérieur de son bloc. Bien qu'elle existe toujours une fois sorti du bloc, elle n'est plus utilisable.

Le gros avantage de cette classe est sa rapidité : lors de l'exécution, le programme n'a pas besoin de chercher la variable ni de réserver de la place en mémoire pour elle, c'est le compilateur qui s'est chargé de cela une fois pour toutes. L'inconvénient, corollaire de cela, est que la place justement réservée ne peut pas être libérée, sauf en quittant le programme.

Tout comme la classe `auto` pour les variables globales, une variable statique globale n'est toutefois visible (et n'existe) qu'à l'intérieur d'un module (un fichier source). Ne confondez pas l'attribution de la zône mémoire avec la visibilité : une variable de type `static` est située en mémoire tant que le programme existe, mais elle suit les mêmes règles de visibilité que les variables de classe `auto` : elle n'est utilisable qu'à l'intérieur de son bloc de localité.

Pour preuve, vous pouvez essayer de compiler l'exemple suivant :

```
/**********************************/
/* programme exemple classe static */
/*                                 */
/* n'est pas compilable !          */
/**********************************/
main()
{
  static short a=2;
  auto   short b=2;
  printf("a=%d, b=%d\n",a,b);
  autre();
}
autre()
{
  auto  short b=3;
  printf("a=%d, b=%d\n",a,b);
}
```

extern

Cette classe ne réserve absolument aucune place en mémoire et ne crée aucune variable. Son rôle, au début d'un module, est d'indiquer au compilateur que la variable est déclarée et créée dans un autre module, qui sera linké plus tard avec l'actuel. Le compilateur reconnaît ce fait, et permet de travailler avec la variable comme si elle existait. Lors du link, le lien (traduction du mot link) est fait entre les deux.

L'avantage de cette classe est de permettre à plusieurs modules de partager les mêmes variables globales. Mais on évitera de l'utiliser, tout comme on doit de préférence éviter les variables globales à un module. En effet, si une fonction d'un module utilise une variable globale, cela pose déja un problème de lisibilité et de clarté de cette fonction. Si en plus la variable provient d'un autre module, elle n'est même pas créée ni initialisée dans le module en question, donc en réalité on ne sait pas vraiment d'où elle provient, ni les valeurs qu'elle pourrait prendre, etc. On dit qu'elle facilite les effets de bord.

Toutefois, dans certains cas cette classe peut rendre de grands service, par exemple si le fichier source d'un module est trop long et qu'il faut le couper en deux modules, on recopiera les déclarations de variables globales du premier morceau pour les intégrer en `extern` dans le second. Attention, on ne peut pas initialiser une variable externe.

Enfin, autre cas où cette classe peut être utile : si plusieurs modules sont utilisés par un module principal et travaillent sur un gros tableau de données, par exemple. Dans ce cas le seul moyen de ne pas recréer ce tableau pour chaque module, c'est d'utiliser un tableau global dans le module principal, déclaré `extern` dans les sous-modules.

Attention encore une fois : une telle méthode rend les modules dépendants du module comportant la déclaration et la définition originale. On perd l'indépendance des modules, qui doivent alors obligatoirement être linkés avec

un module contenant la création des variables `extern`. Théoriquement, un module doit pouvoir être linké avec n'importe quoi, et doit donc être totalement indépendant. Si l'on a besoin de variables globales, elles seront automatiques afin de limiter leur visibilité à ce module. N'oubliez pas qu'une variable de classe `extern` n'existe pas dans le module qui la déclare : elle n'y est pas définie à proprement parler, un autre module doit forcément la définir par ailleurs.

register.

Cette classe de variable fait couler beaucoup d'encre ! Elle ne peut être utilisée que pour les types entiers. Son rôle est d'indiquer au compilateur que, si c'est possible au moment de la création de la variable, il faudra de préférence utiliser les registres du processeur plutôt qu'une zone mémoire pour la stocker.

Cette classe , théoriquement, permet d'optimiser la vitesse de traitement si on la restreint à des variables trés utilisées. Toutefois, le nombre de registres disponibles ainsi que leur taille jouent un rôle important : sur un IBM-PC, avec Turbo C, on peut parfaitement définir une variable `register long` mais ce sera sans effet car les registres du processeur ne peuvent pas contenir 4 octets.

Inversement, si on déclare une register `char`, on a aucune chance que cette variable soit effectivement placée dans un des registres 8 bits du 8086 ou 286/386, car Turbo C n'admet que des variables 16 bits. Il faut obligatoirement que la variable soit de type `int` ou `short` (16 bits), et seul deux registres du processeur sont disponibles. Si l'allocation d'un registre du processeur n'est pas possible, la variable sera implicitement de type `auto`, il n'y a donc aucun problème. Le compilateur assigne les deux registres aux variables de classe `register` les plus utilisées dans le bloc, les autres sont placées en mémoire avec une classe automatique. Vous pouvez en déclarer autant que vous voulez : seules les deux plus utiles seront effectivement rangées dans les registres.

Notez que, paradoxalement, cette classe de variable pourtant très liée au processeur et à la machine utilisée, ne pose absolument aucun problème de portabilité puisqu'elle est ignorée si l'affectation d'un registre n'est pas possible (ceci sur toutes les machines, et par tous les compilateurs).

Enfin, si cette classe vous pose un réel problème de compréhension, oubliez la, elle n'est en aucun cas indispensable ! Mais elle est une parcelle de la face visible de l'iceberg : C autorise un contrôle beaucoup plus profond de la machine et du processeur. Mais selon l'expression consacrée, ceci est une autre histoire, et qui n'a pas sa place dans cet ouvrage (ou si peu !).

RESUMONS NOUS

Ceci termine les classes de stockage disponibles. N'oubliez pas que vous pouvez dans tous les cas ignorer purement et simplement cette partie de la définition des variables. Retenez juste ceci :

- La classe `auto` est la classe par défaut de toute variable, quel que soit son type. Elle indique une création et un stockage dynamique des variables lors de l'exécution et fonctionne dans tous les cas.

- La classe `static` force la réservation de mémoire lors de la compilation plutôt que lors de l'exécution.

- La classe `extern` permet de récupérer les variables d'autres modules sans leur réserver de place. A éviter.

- La classe `register` permet, sous Turbo C et sur IBM PC, de stocker les deux variables `int` ou `short` les plus utilisées dans les registres 16 bits du processeur pour optimiser leur temps d'accès. Cette opération est totalement transparente.

Dans tous les cas, la classe `auto` (donc par défaut) convient à tous les problèmes. Les autres classes ne sont là que pour faciliter un contrôle plus poussé de la réservation mémoire et des accès aux variables.

LES TYPES DERIVES

Turbo C, et le langage C en général, autorise plusieurs types de données qui sont équivalents aux types simples vus plus haut, mais utilisés d'une façon différente. Les tableaux sont un exemple de ces types dérivés : ils constituent en réalité une façon différente d'accéder à un ensemble de variables d'un certain type, mais ils sont bel et bien caractérisés par les mêmes règles que celles de ce type. Les types dérivés en C sont les suivants :

- le type `enum`, qui correspond aux types utilisateur énumérés en Pascal, permet de construire un type de données personnalisé contenant des valeurs entières mais associées à des noms symboliques;

- le type `struct` équivaut au `record` du Pascal et permet de regrouper un ensemble de variables de types différents sous une même appellation;

- le type `union` est presque équivalent à `struct`, mais n'autorise le stockage simultané que d'une seule des variables de l'ensemble. Nous nous pencherons sur les `unions` en détail car elles représentent une grande puissance;

- les pointeurs que nous avons déjà vus sont en réalité des entiers spéciaux (de 2 ou 4 octets suivant le cas) qui contiennent l'adresse d'autres variables. Nous ne les évoqueront pas de nouveau ici, sauf pour mémoire.

- les tableaux que nous avons déjà examinés en détail sont également des types dérivés du type de leurs cases.

Nous allons maintenant passer les types dérivés en revue.

Le type enum

Ce type permet de créer un type de données personnalisé. Un exemple va clarifier les choses :

```
main()
{
  enum classe {STATIC,AUTO,REGISTER,EXTERN};
  enum Dtype{INT,CHAR,SHORT,LONG,DOUBLE,FLOAT,STRUCT,UNION,ENUM}
  enum classe variable=STATIC;
}
```

Dans cet exemple purement démonstratif, nous avons créé deux types énumérés. Nous avons tout d'abord le type `classe`, qui peut prendre les valeurs STATIC, ou AUTO, etc. Notez que les valeurs possibles sont indiquées en MAJUSCULES, pour deux raisons. Tout d'abord, le type ainsi défini est en réalité un type `short`, car en vérité STATIC vaut 0, AUTO vaut 1, et ainsi de suite. C'est ainsi que les valeurs sont stockées. La définition de ces noms équivaut donc à :

```
#define STATIC 0
#define AUTO 1
```

et aisi de suite. Comme il s'agit en quelques sortes de constantes redéfinies, nous leur donnerons toujours par convention des noms en MAJUSCULES. Et la seconde raison, beaucoup plus explicite ici, c'est que les mots `static`, `auto` et autres en minuscules sont réservés, et appartiennent au langage. Nous ne pouvons donc pas les utiliser ainsi.

Le deuxième type est analogue au premier : INT sera stocké sous forme d'un `short` ayant la valeur 0, CHAR prendra la valeur 1, etc. Enfin, nous avons ensuite la déclaration d'une variable de type `enum classe`. Cette variable peut prendre l'une des valeurs indiquées dans le type énuméré qui lui correspond. Dans notre cas, nous lui assignons la valeur STATIC.

Une variable de type énuméré peut être utilisée de la même façon qu'une variable de type `short` : par exemple dans une boucle, en indice d'un tableau de même type, etc.

Le programme suivant met en évidence la puissance (d'un strict point de vue programmation informelle, puisqu'on peut obtenir exactement le même résultat uniquement avec des variables `short`) des types énumérés :

```
/*-----------------------*/
/* ENUM.C                */
/* exemple type enum     */
/*-----------------------*/

main()
{

  /* -------------- type énuméré tricolore --------------- */

  enum  marque  { RENAULT,CITROEN,PEUGEOT };

  /* ---------- quelques constantes pour affichage--------- */

  char  *nom      [PEUGEOT+1] = {"Renault","Citroen","Peugeot"};
  char  *modele [PEUGEOT+1] = {"5","AX","205"};

  /* ------------- UNE variable tricolore --------------- */

  enum  marque indice;

  /* --------------- Le petit programme ! --------------- */

 puts("\n\nBONJOUR, voici notre choix de marques :\n\n");
 for (indice=RENAULT; indice<=PEUGEOT; indice++)
  puts(nom[indice]);
 puts("\n\n ET voici nos modeles :\n\n");
 for (indice=RENAULT; indice<=PEUGEOT; indice++)
  printf("%s %s\t",nom[indice],modele[indice]);
 puts("\n\nCOCORICO\n");
}
```

Vous remarquerez l'utilisation en tant qu'indice de la variable `enum marque indice`. Cet exemple démontre simplement la clarté du listing lorsque l'on utilise des types `enum` : la boucle `for` est très explicite, elle prend les valeurs de `RENAULT` jusqu'à `PEUGEOT`. La variable `nom[PEUGEOT]` nous donnera accès à la chaîne `Peugeot`, etc. De même, notez l'utilisation dans les déclarations de tableaux.

Remarquez que nous pourrions utiliser des nombres à la place des types énumérés : par exemple, le programme suivant effectue strictement les mêmes opérations :

```
/*---------------------------------*/
/* ENUM2.C                         */
/* Exemple de type énuméré simulé  */
/*---------------------------------*/


#define NOMBRE 3

main()
{

  /* ------ quelques constantes pour affichage------------- */

  char  *nom    [NOMBRE] ={"Renault","Citroen","Peugeot"};
  char  *modele [NOMBRE] ={"5","AX","205"};

  /* --------------- Quelques variables ... ---------------- */

  short  indice;

  /* --------------- Le petit programme ! ------------------ */

  puts("\n\nBONJOUR, voici notre choix de marques :\n\n");
  for (indice=0; indice<NOMBRE; indice++)
        puts(nom[indice]);
  puts("\n\n ET voici nos modeles :\n\n");
  for (indice=0; indice<NOMBRE; indice++)
        printf("%s %s\t",nom[indice],modele[indice]);
  puts("\n\nCOCORICO\n");
}
```

Vous obtenez exactement le même affichage. Et pour cause, le programme compilé est identique. Mais le listing n'est pas forcément plus clair, au contraire : la boucle `for` ne nous donne plus aucune indication précise sur ce que nous faisons.

A retenir:

Le type `enum` permet de clarifier de façon spectaculaire les blocs de programme. Il convient d'utiliser systématiquement les majuscules pour les indicateurs du type déclaré, et de se souvenir que le type `enum` peut être utilisé exactement comme un type `short`.

Quelques remarques concluront ce passage sur le type `enum`. Tout d'abord, vous pouvez lors de la définition du type modifier les entiers associés aux identificateurs. Si une initialisation suit un des identificateurs, celui-ci prend la valeur indiquée, sinon il prend celle de l'identificateur précédent, augmentée de 1. Exemple :

```
enum poids_4bits { AUCUN,B0,B1,B2=4,B3=8 };
```

Dans ce type, les identificateurs ont les valeurs suivantes :

```
AUCUN = 0
B0    = 1
B1    = 2
B2    = 4
B3    = 8
```

En effet, les trois premiers (`AUCUN, B0, B1`) sont associés aux valeurs par défaut : 0 pour le premier, 1 pour le suivant, 2 pour le suivant. Si `B2` n'était pas redéfini, il prendrait la valeur 3. Après `"B2=4"`, si `B3` n'était pas redéfini à 8, il prendrait la valeur de `B2+1`, soit 5.

Autre remarque pour les programmeurs Pascal : vous théoriquement bien le type `enum`, puisque l'on dispose de la même facilité en Pascal. En effet, voici un exemple :

EN C :

```
enum ingredient { BEURRE, SEL, POIVRE, OEUF };
```

EN PASCAL :

```
TYPE ingredient = (beurre,sel,poivre,oeuf);
```

Les différences entre C et Pascal à ce niveau sont les suivantes :

- Pour utiliser un ingrédient en indice d'un tableau, il est obligatoire, en Pascal, que celui-ci soit déclaré de la façon suivante:

```
VAR tableau : ARRAY [ingredient] OF ....;
```

Alors seulement vous pouvez utiliser une variable de type `ingredient` en indice dans une boucle pour accéder aux cases du tableau.

En C, l'équivalence entre les types énumérés et le type `short` est totale, cette contrainte n'existe pas. Toutefois on peut contourner cette difficulté en Pascal grace à l'équivalent du "cast" de C. Ainsi, si valeur est un entier, l'expression `ingredient(0)` représentera beurre, etc.. De même, `integer(poivre)` vaudra 2. On peut donc effectuer exactement les mêmes choses qu'en C, mais il faut passer par une conversion qui effectue la traduction du type entier au type énuméré, alors qu'en C l'équivalence est automatique.

- Alors qu'en Pascal standard, un type énuméré ne peut pas recevoir plus de 255 identificateurs (car stocké sur un octet), en C, le type `enum` équivaut à

short et accepte donc 65535 valeurs différentes . C'est une nuance tout à fait théorique (imaginez l'allure du listing avec 65000 identificateurs différents !), sauf en ce qui concerne la remarque ci-dessous.

- La plus importante différence est l'impossibilité, en Pascal, de modifier la valeur entière associée aux identificateurs du type énuméré. vous ne pouvez pas obtenir l'équivalent du type poids_4bits ci-dessus, sauf en rusant. Exemple :

EN C :

```
enum poids_4bits { AUCUN,B0,B1,B2=4,B3=8 };
```

EN PASCAL :

```
TYPE poids_4bits = (AUCUN,B0,B1,nul1,B2,nul2,nul3,nul4,B3 );
```

En effet, les identificateurs nulx occupent des emplacements, et ainsi B2 vaut bien 4 et B3 vaut 8. Avouez que c'est encombrant et pas trés joli. Et surtout, vous ne pouvez pas, à cause de la limite à 255 identificateurs, réaliser en Pascal un type comme celui-ci :

```
enum grand { DIXMILLE=10000,VINGTMILLE=20000 };
```

Quoi que vous fassiez, en Pascal standard, c'est impossible (sauf avec certains compilateurs comme celui de MetaWare, mais il coûte tout de même près de 10000 Francs...). La conclusion de ces remarques, c'est que vous gagnez en facilité d'utilisation des types énuméré en C. Le seul endroit où cela peut être moins clair, c'est dans la déclaration des tableaux, comme nous l'avons vu lors du chapitre Premiers pas. Mais ce n'est pas dû aux types énumérés. Globalement, le type enum correspond tout à fait à l'utilisation classique des types énumérés en Pascal, mais il est plus souple d'emploi en C.

Le type struct

Le langage C possède comme son confrère Pascal un type permettant de regrouper plusieurs variables de types différents sous une même appellation. En Pascal, ce type se nomme record, en C il s'agit simplement de struct, abréviation de structure.

Une structure intègre un nombre tout à fait quelconque de champs. Ces champs peuvent également être quelconques : aussi bien un int qu'un enum ou même un autre struct !

La présentation du type ressemble à ceci:

```
struct blabla
  {
        type1 nom1;
        type2 nom2;
        . . .
  };
```

Les champs possèdent donc un nom qui leur est propre, et sont précisés avec leur type. Prenons un exemple simple, celui d'une structure `adresse` :

```
struct adresse
  {
        short  numero;
        char   rue[30];
        long   code_postal;
        char   ville[20]
  };
```

Grace à cette déclaration, nous pouvons ensuite déclarer, initialiser ou utiliser une variable de type `adresse` de la façon suivante :

```
struct adresse psi={ 5, "Place du Cnl Fabien",75010,"PARIS" };
```

ou

```
struct adresse psi;
psi.numero = 5;
psi.rue    = "Place du Cnl Fabien";
psi.code_postal = 75010;
psi.ville  = "PARIS";
```

L'accès aux différents champs se fait en effet, comme en Pascal, par le nom de la structure, suivie d'un point et du nom du champ. Si le champ est également une structure, on ajoute encore un point et le nom du champ à obtenir, et ainsi de suite.

Voici un exemple de structure intégrée dans une structure et de la déclaration d'une variable initialisée :

```
struct  societe
  {
        char
        nom[20];
        struct adresse   adr;
  } ;
struct societe psi;
psi.nom              = "Editions du PSI";
psi.adr.numero       =  5;
psi.adr.rue          = "Place du Cnl Fabien";
psi.adr.code_postal  =  75010;
psi.adr.ville        = "PARIS";
```

L'utilisation du type `struct` devient très vite une habitude et, fort heureusement, c'est une trés bonne habitude. Mais il y a certaines restrictions à respecter concernant les champs et leur type :

- On ne peut pas préciser la classe de stockage d'un champ. Cela se conçoit puisque les champs sont rangés les uns à la suite des autres en mémoire, et que l'on ne doit avoir aucun contrôle sur cela pour éviter les catastrophes !

On peut par contre bien sûr préciser la classe de stockage d'une variable de type structuré; par exemple dans l'exemple précédent on aurait pu préciser `static struct societe psi`. Evidemment, ce n'est pas très beau ! Nous verrons bientôt comment remédier à cela.

- On doit obligatoirement préciser le type de chacun des membres. Mais il n'y a guère de problème de ce coté là.

- Il est impossible, dans la déclaration d'une structure, de placer un champ qui est du type... de cette structure ! Par exemple, la structure suivante ne peut pas être réalisée :

```
struct AuSecours
 {
       AuSecours CaRepart
 };
```

Il est heureux que l'on ne puisse pas le faire, car en une simple déclaration, on demanderait de réserver de la place pour stocker une variable `AuSecours`, laquelle occupe la place pour stocker un type `AuSecours`, laquelle etc....De plus, il n'y a dans ce type aucune référence à un type connu du C ! C'est totalement inconsistant.

- En revanche, on a le droit d'utiliser le type que l'on définit si l'un des champs est un pointeur (tout comme en Pascal) puisqu'un pointeur est un type connu de C même quand il pointe théoriquement vers un type non encore connu. Ceci permet de réaliser des listes chainées, des arbres binaires, et autres structures évoluées. Par exemple :

```
struct fiston
  {
       struct fiston *papa;
       struct fiston *maman;
       char nom[30];
  };
```

Cette structure, utilisée au maximum de ses possibilités, permet de remonter dans l'arbre généalogique jusqu'à Adam et Eve, à condition bien sûr de le remplir (au delà, nous dirons pour simplifier qu'il n'y a plus assez de place en mémoire). Le langage C mène à tout !

Toutefois, ce genre d'application n'est pas conseillée au débutant, parce que les pointeurs sont ce qu'il y a de plus complexe dans un langage de programmation (et aussi, curieusement, de plus puissant). Si vous voulez vous lancer dans ce type de structure, nous vous conseillons de lire les ouvrages référencés dans l'annexe "bibliographie" avant de tout découvrir par vous-mêmes !

Le type union

Le langage C possède une version spéciale du type `struct`, appelée `union`. Ce type, qui possède lui aussi des champs de types différents, a la particularité de n'en stocker qu'un seul à la fois. En clair, il retient la zone mémoire nécessaire au stockage du champ le plus encombrant, puis il se comporte comme un type `struct`, sauf qu'il n'utilise que cette zône pour tout champ. Un exemple clarifiera les choses. Entrez, compilez et exécutez programme ci-dessous.

```
/*------------------------------*/
/* UNION.C                      */
/* Illustration d'un type union. */
/*------------------------------*/
main()
{
  union  zone
  {
        char   octet;
        short  mot16;
        long   mot32;
  } ;
  union zone test;

  /* modifier MOT16 sans en avoir l'air ! */
  test.mot32 = 0;
  test.octet = 255;
  /* LA PREUVE ! */
  printf("test.mot16 : %d\n",test.mot16);
  /* modifier mot32 idem.*/
  test.mot16 = 12345;
  printf("test.mot32 : %d\n",test.mot32);
  /* modifier octet */
  test.mot32 = 98765;
  printf("test.octet : %d\n",test.octet);
}
```

Le type `union` est utilisé non pas, comme on pourrait le penser, en tant que façon pratique d'économiser de la place -on risque surtout de gagner des erreurs dramatiques en réfléchissant ainsi- mais plutôt pour travailler de façon simple avec des entités proches de la machine. Le type `union` permet par exemple d'accéder à une zône mémoire contenant parfois un octet de données utile, parfois 2, parfois 4. Si le type `union` offre la possibilité d'accéder à ces trois formes, on évite de procéder à des opérations arithmétiques peu lisibles.

Le type `union` est souvent utilisé pour servir d'interface avec des routines programmées directement en assembleur et intégrées au code généré par Turbo C, ou pour appeler les fonctions internes de MSDOS, etc.

En ce qui concerne les applications générales, il vaut mieux éviter l'utilisation de ce type, mais il est intéressant de connaître son existence pour plus tard (lorsque vous en viendrez à programmer plus près de la machine) car il récèle une grande puissance. C'est l'exemple même de l'outil à ne pas mettre entre toutes les mains !

LES TYPES DEFINIS PAR L'UTILISATEUR

Nous avons vu que C permet d'assembler des types de données de base par le type `struct`. Mais la déclaration reste peu sympathique, pour ne pas dire illisible. C'est pourquoi il existe une instruction de déclaration plus astucieuse qui nous permet de créer littéralement un nouveau type de données, et non plus uniquement un assemblage des types de bases.

typedef

`typedef` est cette instruction.

Sa syntaxe est la suivante :

```
typedef type NOM;
```

Le type peut être absolument quelconque : aussi bien un type de base qu'une grosse structure ou union. Par convention on utilise les majuscules pour représenter un type défini par `typedef`. Voici quelques exemples :

```
  typedef struct
   {
         char  nom[20];
         char  prenom[20];
   }     PERSONNE;
typedef short    MOT16;
typedef long     MOT32;
typedef char     OCTET;
typedef int            BOOLEEN;
typedef unsigned short COMPTEUR16;
typedef unsigned long int COMPTEUR32;
```

Et leur utilisation dans des déclarations de variables :

```
#define VRAI -1
#define FAUX 0
main()
{
  PERSONNE       toto;
  COMPTEUR16     i,j;
  COMPTEUR32     memoire;
  OCTET          b;
  BOOLEEN        je_me_demande;
  ...
}
```

Notez qu'outre l'avantage d'éclaircir les déclarations des variables, l'utilisation de `typedef` permet de connaître facilement la taille d'une variable grâce à l'opérateur `sizeof`. Exemples :

```
taille_personne = sizeof(PERSONNE);

taille_compteur = sizeof(COMPTEUR32);
```

L'opérateur `sizeof` est extrêmement utile pour les allocations dynamiques de mémoire (fonction `malloc()`) et les travaux sur les fichiers.

Le type struct pointé

Le type `struct` possède une variante très puissante : la structure pointée. Il ne s'agit pas à proprement parler d'une structure à part entière, mais son utilisation est suffisamment originale pour qu'on y consacre quelques mots.

Le principe est d'accéder à une structure par un pointeur. Par cette méthode, on peut créer des listes chaînées, des arbres binaires, ou autres structures de données assez évoluées et dynamiques. Ceci permet aussi de passer une variable structurée en paramètre à une fonction afin de la modifier. Comme d'habitude avec les pointeurs, il faudra disposer des éléments suivants pour créer une variable structurée pointée :

1. Un type `struct`.
2. Une zone mémoire réservée au stockage de la variable.
3. Un pointeur de ce type qui pointe cette zone.

Voici un exemple concret :

```
#include <stddef.h>
#include <alloc.h>
#include <string.h>

main()
{
  typedef struct
  {
        char *nom;
        char *adresse;
  } PERSONNE;

  PERSONNE        *ptr;
  ptr = (PERSONNE*) malloc(sizeof(PERSONNE));
  if (ptr == NULL)
  {
        printf("IMPOSSIBLE DE RESERVER DE LA MEMOIRE !!!\n");
        exit(1);
  }
  strcpy(ptr->nom,"Je suis TOTO.");
  strcpy(ptr->adresse,"J'habite chez moi !");
  printf("Contenu : \n%s\n%s\n\n",ptr->nom,ptr->adresse);
}
```

CHAPITRE 13

L'ARITHMETIQUE DODUE

Comme nous l'avons laissé entendre, le langage C est trés puissant au niveau du traitement des expressions. En réalité, il est extrêmement facile d'assembler des expressions et des opérations de calcul en C, mais on obtient tout aussi facilement des choses illisibles.

Dans ce chapitre, nous allons nous pencher sur l'utilisation des expressions de façon plus poussée. Notre but sera également de vous faire adopter de bonnes habitudes, c'est à dire apprendre à éviter les mauvaises.

Ce chapitre paraîtra certainement redondant à certains d'entre vous : notre but est ici de mettre en évidence certaines des particularités importantes ou sournoises du langage C. Il est vital que vous maîtrisiez correctement les notions qui suivent avant de poursuivre l'étude du langage.

Si l'on excepte l'existence des instructions comme `switch`, `for`, `else`, le langage C comporte tous les éléments nécessaires pour créer des listings totalement abstraits et illisibles.

Prenons un exemple simple. Le programme ci-dessous affiche la liste des carrés des nombres de 1 à 20.

```
main()
{
   int   i=0;
   while  (20-i)
         if  (i!=20)
                printf("%2d  %4d\n",i++,i*i);
}
```

Il n'est pourtant pas compliqué, mais il est déjà difficilement lisible. Et ce n'est pas qu'une question de présentation. Le programme comporte d'autre part un test dangereux, une redondance et une instruction dont la place est discutable.

Le test dangereux est l'égalité. En effet, l'expression `(20-i)` testée dans le `while()` ne devient fausse que si `i` arrive à 20. Ce n'est pas trés grave ici car `i` est de type `int`, mais c'est le genre de test à éviter avec les nombres flottants car il y a toujours une décimale trainante qui perturbe les tests d'égalité.

Il est donc plus prudent de tester `(20>i)`. Ce test fonctionnera aussi bien pour des entiers que pour des réels.

La redondance : le test `if` ne sert à rien puisque la boucle s'arrête lorsque `i` atteint 20.

L'instruction mal placée : `i++` devrait se situer à l'extérieur de l'instruction `printf`, car suivant le compilateur, `i++` ou `i*i` sera exécutée en premier, ce qui provoquera une différence de fonctionnement.

De plus, dans un tel cas, il est beaucoup plus judicieux d'utiliser une boucle `for`. L'exemple suivant est correct :

```
main()
{
   short i ;

   for (i=1 ; i<20 ; i++)
        printf("%2d %4d\n",i,i*i);
}
```

Nous n'avons pas donné cet exemple pour rien, mais juste pour montrer comment une même tâche peut être programmée de deux façons totalement différentes, l'une dangereuse, l'autre propre et lisible. Inutile de préciser laquelle doit être adoptée.

Dans toute programmation en C, le même contraste peut se retrouver.

On aura très souvent, pour ne pas dire toujours, le choix entre une multitude de possibilités pour programmer une tâche donnée. Il faut de préférence choisir en fonction d'un critère bien précis, et rester homogène dans cette attitude pour tout le programme :

- Le premier critère est la portabilité : dans ce cas, tenir compte des nombreuses remarques à ce sujet. Se contenter de la norme K&R, éviter les ambiguités de chronologie (expressions multiples utilisant une même variable), ne pas utiliser de types de données non standard, etc...

- Le second critère est la performance : suivez les remarques concernant l'optimisation dans le manuel Turbo C, mais acharnez-vous plus à trouver de bons algorithmes qu'à mettre en œuvre des astuces du C.

LES TYPES ENTIERS

Nous l'avons maintes fois souligné, C possède différents types de base compatibles entre eux.

Notamment, et c'est sans doute là que se situent les plus grosses astuces et les plus gros dangers, au niveau des nombres entiers.

En bref, une valeur entière peut prendre l'un des types C suivants :

```
int
short
long
char
unsigned
unsigned short
unsigned long
unsigned char
float
double
```

Voilà qui nous change des possibilités des autres langages. De plus, vous l'aurez deviné, si par exemple une variable `toto` est de type `float`, rien n'empêche de considérer son contenu comme un `char` : c'est l'évaluateur d'expressions du langage C qui effectuera la conversion.

Pour ces raisons de compatibilité, il est nécessaire de travailler avec prudence.

En effet, si le stockage d'un caractère `char` dans une variable `short` ne pose aucun problème, il en va tout autrement dans l'autre sens.

La clé est le nombre d'octets utilisé pour chaque type:

`char`	1 octet
`int`	2 octets
`short`	2 octets
`long`	4 octets

Si la variable `short` contient une valeur comprise entre 0 et 255, elle peut directement être utilisée comme un `char`. Mais si elle se situe à l'extérieur de cet intervalle, le résultat sera sans doute ... n'importe quoi.

Compilez et exécutez le programme suivant pour mieux comprendre ces problèmes.

```
main()
{
   long  i ;

   for  (i=0L; i<50000L; i+=500L)
        printf("char : %c\nshort: %hd\nlong : %ld\n",(char) i,
                (short)i,i);
}
```

Dans cet exemple, vous ne sauriez pas vraiment dire quel est le code du caractère récupéré, ni le nombre de type `short` dès que l'on dépasse 32767, puisqu'il est interprété comme un nombre signé.

Vous devrez donc faire attention à ce que les conversions de type aillent toujours dans le sens croissant du nombre d'octets utilisés.

Voici donc les conversions automatiques sans danger :

```
char   ->  short, long, float ou double
short  ->        long, float ou double
long   ->              float ou double
float  ->                    double
double -> pas de conversion sans danger
```

N'oubliez pas que la majorité des fonctions de la bibliothèque renvoient un résultat de type `int`. Nous avons mis en garde le lecteur contre ce type, car sa taille varie d'un système à un autre. Il vaut mieux utiliser le type `short`.

En ce qui concerne les fonctions de la librairie, bien entendu, il faut impérativement travailler en type `int` pour tester les résultats des fonctions de type `int`. Sans quoi on prend un énorme risque.

La plupart du temps on reçoit en cas d'erreur la valeur `-1`. Et contrairement aux idées reçues, il faut savoir que `-1` ne vaut pas forcément `-1` !

En effet, si la fonction renvoie par exemple une valeur `int` de `-1`, et que vous récupérez ce résultat dans une variable de type `long`, vous obtiendrez `32768` et non pas `-1`. C'est pourquoi vous devez impérativement récupérer le résultat d'une fonction dans une variable de même type avant d'entreprendre le moindre test.

RAPPEL SUR LES EXPRESSIONS

Expression simple :

Une `expression simple` peut prendre la forme suivante :

```
lvalue operateur = expression simple
```

Le malheur de cette notation, c'est qu'elle est récurrente ! En effet, une expression peut en contenir une autre. Nous avions vu qu'une expression, outre le fait de calculer une valeur ou un résultat, possèdait intrinsèquement cette valeur calculée.

Cette particularité cache encore l'un des principaux facteurs de la puissance de C. En effet, considérons la ligne suivante :

```
if ( ungetc( c = getchar() ) == EOF ) )
   puts ("fini !");
else
   printf("%c",c);
```

Voici, dans l'ordre, ce qu'effectue l'expression qui suit le `if`:

- D'abord, appel de la fonction `getchar()`. Cette fonction renvoie une valeur de type `int`. C'est `-1` si la fin du fichier `stdin` est atteinte, sinon c'est le caractère obtenu.

- Ensuite on assigne cette valeur à la variable `c`. L'ensemble possède toujours la valeur ... de cette valeur (!).

- `ungetc()` replace cette valeur dans le tampon de `stdin`. On reçoit `-1` si un problème est survenu (fin du fichier obtenue), `0` sinon.

- En comparant la valeur renvoyée par `ungetc()` à `EOF` , nous savons si c'est la fin du fichier `stdin`.

- Si ce n'est pas la fin nous pouvons afficher `c`.

Nous avons donc évalué, successivement, les valeurs (ou expressions) suivantes :

```
                getchar()
          ( c = getchar() )
 ungetc ( c = getchar() )
(ungetc ( c = getchar() ) == EOF)
```

Remarquez que chacune de ces expressions a une valeur, y compris la dernière qui est fausse (`0`) ou vraie (différente de `0`).

Rappelez-vous que les booléens n'existent pas en C : toute expression différente de 0 vaut `VRAI`, toute expression nulle vaut `FAUX`.

Expression composée

Enfin, rappelons qu'une expression peut être composée : il suffit pour cela de séparer les expressions par des virgules et d'enclore le tout entre parenthèses. Les expressions seront évaluées de gauche à droite une par une.

`(expression1 , expression2, expression3)`

correspond à l'enchaînement de

```
expression1
expression2
expression3
```

et possède la valeur intrinsèque de `expression3`.

N'abusez pas des expressions composées, elles alourdissent énormément les lignes de programme et la lisibilité totale s'en ressent durement. De plus, cela n'apporte pas grand chose.

Expression conditionelle

L'expression conditionelle se présente sous la forme suivante :

`(expression) ? (expression1) : (expression2)`

Et toute cette ligne possède la valeur de `(expression1)` si `(expression)` a une valeur vraie (différente de `0`), sinon elle renvoie la valeur de `(expression2)`.

En bref, cette expression comporte un test, lequel influe sur la valeur de l'expression !

Voici un exemple :

```
i = (i==1) ? 2 : 1;
```

Cette ligne a pour effet de faire prendre à `i` la valeur `2` si `i` vaut `1`, et la valeur `1` si `i` vaut `2`. Il s'agit d'une alternance entre les deux valeurs. En réalité cette ligne équivaut aussi à :

```
if (i==1)
      i=2;
 else
      i=1;
```

Ou encore :

```
i = 2=(i==2);
```

Méditez ces exemples, et n'oubliez pas : une condition vraie a la valeur `1`, une condition fausse a la valeur `0`.

CASTING

Le casting du langage C n'a aucun rapport avec le cinéma, qu'on se le dise.

Il s'agit de l'équivalent du `retyping` du Pascal. C'est ce qui permet au programme de récupérer une donnée d'un type quelconque en modifiant au passage son type.

Le casting se met en œuvre d'une façon extrêmement simple : on fait tout simplement précéder le paramètre, l'expression ou la donnée voulue du nouveau type entre parenthèses. Par exemple :

`(char)65 équivaut à 'A'.`

`(double)1/3`

`équivaut à 1L/3 , donc à 0.3333333333333L`

`(double)(1/3) équivaut à (double)(0) donc à 0L.`

Bien entendu, au niveau de la conversion des nombres, ceci n'a pas grand intérêt car de toutes façons la conversion est automatique (cf la première partie de ce chapitre). Au mieux, on peut avec le casting éviter des problèmes de calcul, comme par exemple pour une division de deux entiers dont on veut garder le résultat sous forme réelle. En effet, si l'on n'utilise pas le casting, dans ce cas, la division d'un entier par un entier est une division entière, et on perd les décimales.

Mais la grande puissance du casting se révèle au niveau des fonctions et des paramètres qui leur sont passés.

L'exemple absolu et le plus utile est la fonction d'allocation mémoire `malloc()` de la bibliothèque. Vous avez déjà pu remarquer son utilité.

`malloc()` reçoit en paramètre unique le nombre d'octets à réserver en mémoire. Elle renvoie ensuite une donnée de type pointeur (l'adresse où se situent les octets qui ont été attribués), et l'on peut alors donner cette valeur d'adresse à un pointeur.

Mais nous avons vu qu'une variable pointeur était forcément déclarée avec son type, justement parce que la zone pointée n'a pas une taille précisée par son adresse. Il faut obligatoirement connaître le type de la donnée pointée pour pointer dessus.

Or `malloc()` ne connaît pas ce type : elle reçoit uniquement comme information le nombre d'octets à réserver.

Il est donc impossible que `malloc()` renvoie un pointeur directement utilisable, puisqu'elle ne connaît pas le type du pointeur à renvoyer.

C'est pourquoi `malloc()` renvoie un pointeur de type `void *`.

`void`, nous l'avons vu, est un type "rien du tout". L'avantage de ce type est que, puisqu'il ne représente rien de particulier, on peut tout lui faire représenter.

Et en effet, c'est la façon dont nous utiliserons la fonction `malloc()` : nous commençons par récupérer un pointeur (type pointeur `void`), et grâce au casting, nous lui attribuons un type correct (par exemple `char *`), ce qui permet alors de récupérer cette valeur dans la variable pointeur.

Voici un exemple :

```
char  *chaine;          /* pointeur de type char */
void  *intermédiaire;   /* pointeur de type "rien du tout" */
/* nous voulons stocker 80 octets dans la chaîne, il faut */
/* réserver cet espace avec malloc()                      */
intermediaire = malloc(80);

/* La réservation s'est elle bien passée ? */
if (intermediaire == NULL)
{
   puts ("Problème mémoire....");
   exit();
}

/* Attribuer le pointeur par casting */
chaine = (char *) intermediaire;
```

Notez que cet exemple peut être contenu en beaucoup moins de lignes, et sans perdre grand chose à la lisibilité, d'autant que nous supprimons le pointeur intermédiaire :

```
char *chaine;        /* pointeur de type char */

if ((chaine = (char *) malloc(80)) == NULL)
{
   puts("Problème ...");
   exit();
}
```

Au risque de nous répéter, c'est la toute puissance du C ...

ARITHMETIQUE DECIMALE ET FONCTIONS MATHEMATIQUES

Le C comporte un nombre impressionant d'opérateurs. Nous en donnons la liste dans l'annexe B, avec leur ordre hiérarchique.

Pour les calculs plus complexes, le fichier en-tête `<math.h>` contient les déclarations pour de nombreuses fonctions mathématiques.

Dans l'ensemble, pour les calculs en nombres réels, nous conseillons vivement de n'utiliser que le type `double`, et non `float`.

En effet, comme nous l'avons souligné, le type `float` n'existe pas : tous les calculs sont effectués en `double`, puis ramenés éventuellement en format `float`. Il est donc évident que l'utilisation des nombres au format `float` ralentit le traitement, au lieu de l'accélérer, puisqu'il comporte les mêmes calculs, plus deux conversions de type !

En revanche, si vous travaillez sur de gros tableaux de matrices, la mémoire ne supportera pas forcément que chaque donnée occupe 8 octets. Dans ce cas, certes, il faut bien se résoudre à utiliser `float`, qui n'occupe que 6 octets.

Nous allons maintenant passer en revue les fonctions mathématiques les plus utiles.

Fonctions de valeur absolue

Ces fonctions différent suivant le type de leur opérande.

abs()

Syntaxe:

```
#include <stdlib.h>
int     n ;
int     resultat;

resultat = abs(n);
```

fabs()

Syntaxe :

```
#include <math.h>
double  x;
double  resultat;

resultat = fabs(x);
```

labs()

Syntaxe :

```
#include <stdlib.h>
long    n ;
long    resultat;

resultat = labs(n);
```

Il est important d'utiliser la fonction appropriée suivant le type de la valeur, sans quoi l'on risque d'obtenirr un résultat extrêmement fantaisiste.

Les Fonctions Trigonométriques.

Toutes les fonctions trigonométriques travaillent avec un paramètre de type `double` et renvoient un paramètre de type `double`.

Syntaxes :

```
#include <math.h>
double   x;
double   resultat;

resultat = sin(x);      resultat = cos(x);
resultat = tan(x);      resultat = asin(x);
resultat = acos(x);     resultat = atan(x);
```

où `x` représente un angle en radians.

Les fonctions d'arrondi

ceil()

Syntaxe :

```
#include <math.h>
double   x;
double   resultat;

resultat = ceil(x);
```

Renvoie l'arrondi à l'entier supérieur (`ceil` pour ceiling, plafond). Attention, le résultat est un entier mais de type double : en fait, c'est un réel sans partie décimale.

floor()

Syntaxe :

```
#include <math.h>
double   x;
double   resultat;

resultat = floor(x);
```

Renvoie l'arrondi à l'entier inférieur (`floor` signifie plancher). Attention, le résultat est un réel, sans partie décimale.

Logarithmes, exponentielles et puissances.

log()

Syntaxe :

```
#include <math.h>
double   x;
double   resultat;

resultat = log(x);
```

Renvoie le logarithme népérien de `x`.

exp()

Syntaxe :

```
#include <math.h>
double   x;
double   resultat;

resultat = exp(x);
```

Renvoie l'exponentielle de `x`.

log10()

Syntaxe :

```
#include <math.h>
double   x;
double   resultat;

resultat = log10(x);
```

Renvoie le logarithme en base 10 de `x`.

pow()

Syntaxe :

```
#include <math.h>
double   x,y;
double   resultat;

resultat = pow(x,y);
```

Renvoie `x` à la puissance `y (POWer)`.

sqrt()

Syntaxe :

```
#include <math.h>
double   x;
double   resultat;

resultat = sqrt(x);
```

Renvoie la racine carrée (SQuare RooT) de `x`.

Aléatoires

srand()

Syntaxe :

```
#include <stdlib.h>
unsigned  nombre;

srand(nombre);
```

Initialise le générateur de nombre aléatoires avec une valeur non signée (positive) de 2 à 65535. Si on envoie le paramètre 1, le générateur est réinitialisé avec la valeur initiale (pour générer la même suite de nombres aléatoires).

rand()

Syntaxe :

```
#include <stdlib.h>
intresultat;

resultat = rand();
```

Renvoie un entier signé compris entre -32768 et +32767.

CHAPITRE 14

ENTREES/SORTIES ET FICHIERS

ENTREES ET SORTIES STANDARD DOS ET C, REDIRECTION

En langage C, tout ce qui traite des entrées et sorties est implémenté par des fonctions de la librairie. Comme nous l'avons souligné plusieurs fois, C ne possède aucun instruction d'entrées/sorties.

D'autre part, C a été créé en même temps que le système d'exploitation Unix, et pour lui. Or celui-ci a une particularité intéressante, qui a été reprise sur MS-DOS par Microsoft : la redirection des entrées/sorties est possible.

Qu'est-ce que la redirection ?

Une entrée, pour DOS ou un programme, est un "canal" de dialogue par lequel il peut recevoir des informations ou des données. Il peut s'agir du clavier, bien entendu, mais aussi d'un fichier ou d'un port de communication, par exemple. Tout dispositif matériel ou logiciel est finalement capable de transmettre ou de produire des données. En C et sous MS-DOS, l'ensemble de tous les dispositifs ainsi capables de fournir des données est regroupé sous le terme de *dispositif d'entrée*, et toutes les fonctions chargées de récupérer des entrées savent fonctionner indifféremment avec l'ensemble de ces dispositifs !

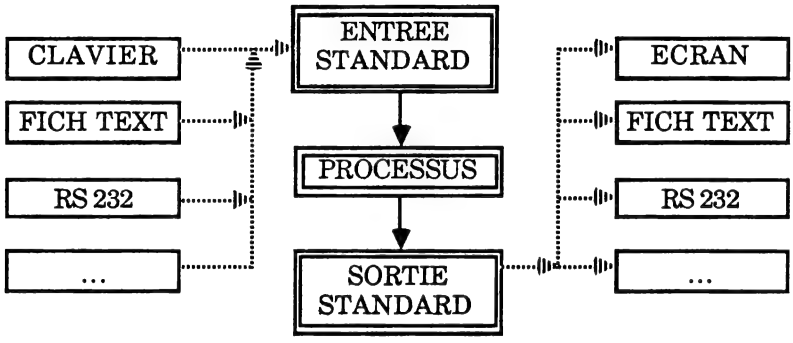
Cela signifie, par exemple, que votre programme pourra traiter des saisies au clavier, et qu'il ne sera pas nécessaire de modifier le source pour qu'il puisse lire les mêmes données dans un fichier, ou les recevoir sur une ligne de communication série.

Ce principe de redirection existe aussi au niveau des sorties : tout dispositif capable de recevoir des données (les sorties) est similaire au niveau du C, et on peut donc trés simplement envoyer les résultats d'un calcul sur l'écran ou dans un fichier suivant le désir de l'utilisateur.

En C et en Unix, il existe trois dispositifs spéciaux appelés entrée standard, sortie standard et erreur standard. Il s'agit des canaux sur lesquels travaillent les fonctions du langage, et par lesquels le systême d'exploitation reçoit ou fournit les données, et envoie les messages erreur.

Anciennes versions du DOS ou BASIC

UNIX/MS-DOS avec C



Sous MS-DOS et avec Turbo C, c'est exactement identique. L'entrée standard par défaut, par exemple, sous MS-DOS, est le clavier, mais on peut trés simplement le remplacer par autre chose grâce à l'opérateur "<" suivi de l'identification d'un dispositif fournissant des données.

C'est par exemple le cas de l'utilitaire MORE du DOS. Celui-ci a pour rôle de recevoir des lignes de texte sur l'entrée standard, et de les envoyer sur la sortie standard en marquant une pause à chaque page écran.

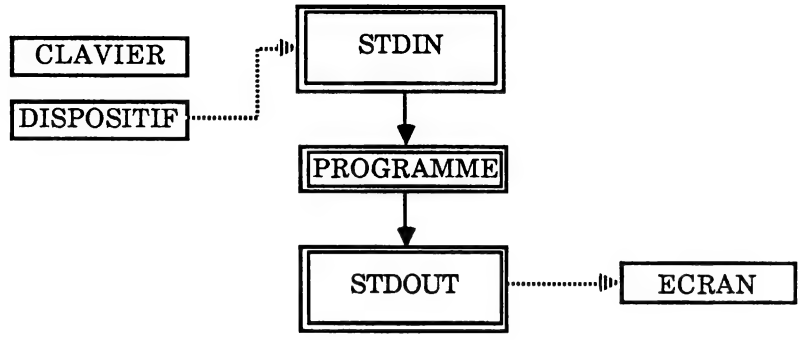
En tapant sous MS-DOS, la commande suivante :

```
MORE <FICHIER.TXT
```

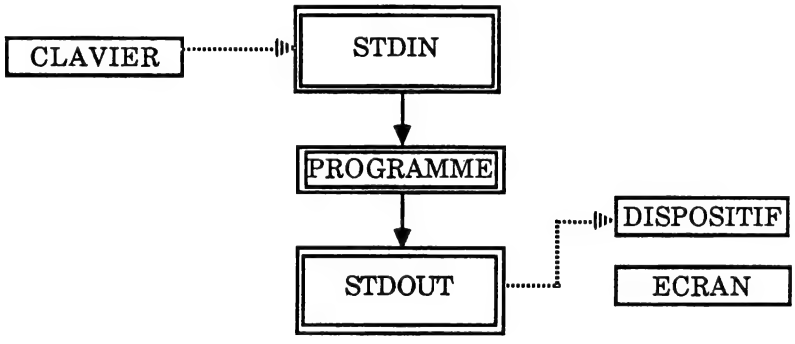
on fournit chaque caractère du fichier FICHIER.TXT (y compris ceux qui marquent les fins de lignes) au programme MORE, via l'entrée standard, qui se redirige sur le fichier et oublie un instant le clavier.

La sortie standard fonctionne de façon similaire, sauf que l'opérateur de redirection est ">". Dans ce cas, MS-DOS oublie un instant l'écran et envoie chaque caractère sur le dispositif de sortie rattaché à la sortie standard.

Redirection des entrées



Redirection des sorties



Bien entendu, si le programme accède au clavier ou à l'écran sans passer par les fonctions internes de MS-DOS (mises à la disposition du programmeur à cet effet), la redirection des entrées/sorties ne fonctionne absolument pas !

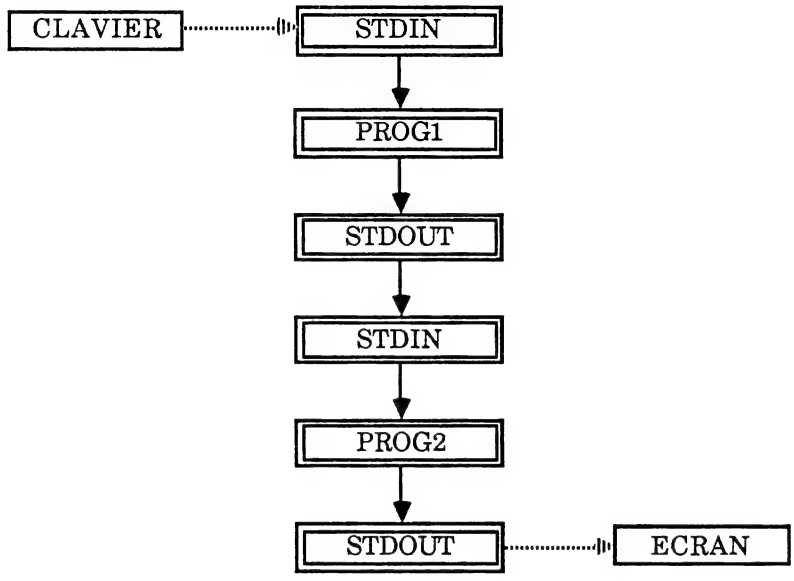
Si vous utilisez les fonctions `scanf` et `printf` du C pour accéder au clavier ou à l'écran, votre programme pourra fonctionner exactement comme l'utilitaire MORE. Il vous suffira de préciser, le cas échéant, sous DOS et en lançant le programme, la nouvelle entrée ou sortie standard.

Enfin, une autre possibilité issue d'Unix existe également sous MS-DOS : le principe du "piping" (prononcer "paille-ping"). En Anglais, le "pipe" est un tuyau. Ce principe d'appel des programmes permet d'enchaîner plusieurs outils en fournissant les sorties d'un d'entre eux comme entrées au suivant, et ainsi de suite. Les programmes sont alors assemblés comme une suite de tuyaux, avec une entrée standard au début, une sortie standard à la fin, et des raccords entre chaque programme.

Le "piping" est mis en oeuvre avec le caractère spécial " | ", souvent appelé pour cette raison "pipe", bien que son appelation ANSI soit "barre verticale".

Le schéma ci-dessous illustre de façon plus claire le principe du piping, que nous utiliserons tout à l'heure. Notez que cette manipulation peut parfaitement être simulée par les redirection > et < en passant par un fichier temporaire.

Principe du 'PIPE'



La commande :

```
PRG1 | PRG2
```

équivaut à la suite de commandes ci-dessous :

```
PRG1 >TOTO.TXT

PRG2 <TOTO.TXT

DEL TOTO.TXT
```

Mais c'est tout de même plus sympathique et moins encombrant avec le piping, il faut l'admettre.

Les entrées et sorties standards : la base de C.

Il faut garder présent à l'esprit le fonctionnement par E/S standard : d'une part il est la clef de l'évolutivité de vos programmes (qui sont en effet garantis compatibles avec les futures versions du DOS par Microsotf si vous utilisez ces fonctions), d'autre part il décuple les possibilités d'un programme. On peut par exemple placer toutes les saisies clavier dans un fichier, initialiser le port série, et lancer le programme pour qu'il s'exécute tout seul et envoie les résultats à un

autre ordinateur, ceci sans modifier la moindre instruction du programme ni le recompiler, alors qu'il était initialementprévu pour recevoir des données tapées au clavier et afficher ses résultats sur l'écran.

En réalité, c'est beaucoup plus qu'un gadget. La redirection des entrées/sorties standards permet aux programmes en langage C d'accéder aux résultats d'un autre programme, d'une commande DOS, ou de fournir des données à celles-ci, etc...

C'est donc l'idéal pour construire la majorité des applications et outils système. C'est bien dans cet esprit que fut conçu C : n'oubliez pas que le système d'exploitation Unix, si l'on excepte quelques centaines de lignes en assembleur, est intégralement écrit en langage C, de même que la majorité des outils qui constituent son environnement.

Enfin, C permettant de récupérer les entrées et sorties standards du DOS, et disposant par ailleurs de nombreuses fonctions d'accès au système (nous les examinerons brièvement dans le chapitre 15), il faut constater que vous pouvez, avec un peu d'l'imagination et d'expérience dans le langage, recréer en C tous les utilitaires de MS-DOS, ou changer radicalement son comportement.

C'est là que réside l'un des principaux attraits du langage C : il constitue un outil parfait de modification d'un DOS.

La redirection : polyvalente mais pas universelle !

Mais attention, le concept de la redirection des E/S admet des limites, notamment celles du bon sens : n'allez pas imaginer qu'on puisse changer la couleur d'un fichier, par exemple. Il est évident que si vous utilisez une instruction de changement de couleur, celle-ci n'aura aucun effet sur le contenu du fichier qui reçoit les caractères de la sortie standard.

De même, il est bien évident qu'un fichier ne peut pas contenir un appui simultané des touches `Ctrl`, `Alt`, et `Del` : il sera donc impossible de placer un RESET sur l'entrée standard, car aucune touche du clavier ne représente une interruption.

Dans l'ensemble, vous devez considérer que l'entrée et la sortie standard fonctionnent exactement comme un fichier texte. Cela signifie que vous pouvez recevoir ou envoyer ... du texte, rien d'autre. D'ailleurs, il n'existe aucun moyen évident d'exécuter une suite de touches tapées au clavier comme s'il s'agissait d'un programme pour le processeur, et la même remarque vaut pour les caractères envoyés à l'écran.

A titre d'exemple, voici un petit programme que vous devrez compiler puis exécuter sous DOS. Nous verrons l'utilisation de la fonction `getchar()` plus tard, mais ce programme vous montre déjà la puissance et la souplesse des entrées/sorties standard. Vous remarquerez que le programme ne procède à aucune gestion de fichier, de clavier ou de variable système : il lit simplement des caractères sur `stdin` et les envoie d'autres sur `stdout` !

```
/************************************************/
/* NUMEROTE.C                                   */
/*                                              */
/* Exemple d'utilisation de la redirection des  */
/* entrées et sorties standard via un programme */
/* en Turbo C. Ce programme reçoit les caractères */
/* sur l'entrée standard et les envoie sur la   */
/* sortie standard en numérotant chaque ligne.  */
/*                                              */
/* Les exemples suivants peuvent ensuite être   */
/* exécutés sous MS-DOS :                       */
/*                                              */
/*  C>NUMEROTE                                  */
/*    puis taper du texte avec des <ENTER>. Pour */
/*    finir, taper un Ctrl-Z ou F6, puis <ENTER>. */
/*                                              */
/*  C>DIR | NUMEROTE >ESSAI.TXT                 */
/*  C>TYPE ESSAI.TXT                            */
/*                                              */
/************************************************/

/*************************************************************/
/* Nécessaire pour obtenir l'accès aux fonctions E/S utilisées.*/
/*************************************************************/

#include <stdio.h>

/*************************************************************/
/* Quelques constantes pour avoir un programme plus lisible  */
/*************************************************************/

#define FIN_LIGNE     '\n'        /* Fin de ligne  * /
#define FIN_FICHIER   '\x1A'      /* Fin de fichier */
                                  /*  = Ctrl-Z     */

/*************************************************************/
/* Prototype de la fonction secondaire d'écriture de caractère.*/
/*************************************************************/

short ecris(char c);

/*************************************************************/
/* fonction principale, travaillant sur l'entrée et la sortie */
/* standards (par défaut, clavier et écran).                 */
/*************************************************************/
```

```
main()
{
  char  c;                          /* Pour l'entrée standard */
  short  ligne = 0;                 /* Compteur de lignes */
  short  vide = 1;                  /* Drapeau ligne vide */

  while ((c = getchar() ) != EOF)   /* Lire un caractère */
  {
    switch(c)
    {
      case FIN_LIGNE:               /* Il s'agit de la fin d'une */
                                    /* ligne, la traiter */
          ecris(c);
          vide = 1;                 /* Drapeau ligne vide. */
          break;                    /* Fin du traitement de fin
                                       de ligne*/

      default:                      /* C'est un autre caractère, */
                                    /* nous pouvons l'écrire.      */
          if (vide)                 /* La ligne était-elle vide ? */
          {                         /* Oui : elle ne l'est plus ! */
            vide = 0;
            printf("%3d: ",++ligne);
          }
          ecris(c);                 /* Dans tous les cas écrire */
          break;
    }
  }
  ecris(FIN_FICHIER);               /* Envoyer signal de fin sur */
                                    /* la sortie standard */
  exit(0);                          /* Sortie avec code succès DOS  * /
}

/*************************************************************/
/* Ecrire un caractère sur le dispositif de sortie standard. */
/* Si l'écriture échoue, envoyer un message sur la sortie    */
/* erreur standard et terminer le programme.                 */
/*************************************************************/

short ecris(char c)
{
  if( (putchar(c) == EOF) && (c != FIN_FICHIER) )
  {
    fputs("Disque sature !!!!",stderr);
                        /* Sortie avec code ERREUR DOS.*/
                        /*fputs place une chaine dans un fichier. */
    exit(1);
  }
}
```

METHODES DE GESTION DES FICHIERS (FCB et handles)

Nous avons vu que C comportait trois dispositifs standards de type fichier : l'entrée `stdin`, la sortie `stdout` et la sortie des erreurs `stderr`.

La méthode du FCB

Sous MS-DOS, un fichier peut être créé et géré de deux façons différentes. La première en lui associant un "bloc de contrôle", ou FCB. Cette méthode assez ancienne suppose que ,quelque part en mémoire, le compilateur réserve une zone pour y placer les renseignements importants comme le nom du fichier, son type d'enregistrement, sa taille, son emplacement sur le disque, etc.

La méthode FCB est commune pratiquement à touts les systèmes d'exploitation, notamment Unix, CP/M et MS-DOS. Tous ces systèmes ont la possibilité de remplir - sur demande de l'utilisateur - un FCB d'après les renseignements fournis sur un fichier. Ensuite, toute opération est demandée en fournissant en paramètre au DOS ou à la fonction d'E/S l'adresse de ce fameux FCB . C'est une méthode assez lourde car le programme utilisateur doit connaître les caractéristiques du fichier. De plus elle ne supporte pas les chemins d'accès (non disponibles sur la version 1 de MS-DOS).

La méthode du handle

L'autre méthode, puisqu'il en existe une seconde, est beaucoup plus moderne. Elle part du principe qu'un fichier est entirèrement caractérisé par son nom complet, et que le programme n'a pas à se préoccuper du reste. De plus, nous l'avons vu, avec les redirections d'Entrées /Sorties nous n'avons pas forcément besoin d'un FCB, et le FCB risque justement de rendre l'accès au fichier dépendant d'un dispositif physique.

C'est pourquoi, dans un but de polyvalence, MS-DOS offre la possibilité de gérer les Entrées et les Sorties par ce que l'on appelle des *handles*.

"Handle" signifie, en bon Français, "poignée". Evidemment ce n'est pas très explicite. On le traduit généralement par descripteur de fichier, ou numéro d'accès. Mais comme aucun terme standard n'existe pour le traduire, et que de nombreux ouvrages utilisent directement le mot "handle", nous allons nous conformer à cette convention. Vous trouverez donc le mot "handle" dans la suite de ce texte. A partir du moment où vous vous intéressez au C, de toutes façons, vous vous exposez à croiser ce terme un jour ou l'autre. Autant le connaître dès maintenant.

Un handle est tout simplement un numéro. Ce numéro peut être associé à un dispositif d'E/S. Toutes les opérations (écriture, lecture...) peuvent faire référence uniquement à ce numéro, sans se soucier de FCB et de zone mémoire réservée. C'est le DOS qui gère la correspondance entre le numéro et ce qu'il représente. Le programme se contente de réclamer un handle en indiquant quel est le nom du dispositif qu'il désire lui associer.

Cette méthode de gestion par le handle est encore une fois directement inspirée d'Unix et n'existe pas sous CP/M. Il n'est donc pas surprenant que nous retrouvions les deux méthodes dans le langage C : celle des FCB, et celle des handles.

QUE CHOISIR?

En effet, la librairie C fournit deux ensembles de fonctions pour gérer les fichiers et les Entrées/Sorties : le premier ensemble travaille avec un type de données qui se nomme `FILE`. Il s'agit d'une méthode similaire à celle des FCB : la structure `FILE` représente en quelque sorte le bloc de contrôle du fichier. Le second ensemble travaille avec des fonctions qui renvoient ou traitent des paramètres de type `int`. Il s'agit du handle, bien entendu.

Vous devez maintenant réaliser, c'est important, que d'un strict point de vue formel, un fichier ouvert par une fonction de type `FILE` est physiquement indentique au même fichier ouvert par une fonction de type `int`.

La méthode utilisée n'intervient pas dans le contenu du fichier !

Mais attention : les deux méthodes sont incompatibles entre elles. En clair, un handle est attribué par l'ouverture d'un fichier, et est libéré par sa fermeture. Pour les FCB, c'est la même chose. Mais en aucun cas vous ne pourrez écrire grâce à un handle sur un fichier ouvert par un FCB. C'est totalement incompatible, même si finalement l'effet est le même.

Pour plusieurs raisons, nous conseillons vivement d'utiliser la méthode des handles et les fonctions `int`. En effet, cette méthode est beaucoup plus simpler car dans tous les travaux vous ne devez plus vous souvenir que d'un simple numéro de type entier. C'est souple et facile à gérer.

D'autre part, le handle est un concept beaucoup plus moderne, et Microsoft recommande son utilisation plutôt que celle des FCB, pour trois raisons essentielles :

a) Cela rend les fichiers totalement indépendants des dispositifs physiques et logiciels (machine et système d'exploitation).

b) MS-DOS (à partir de la version 2) est capable de distinguer de façon trés précise les erreurs (les codes erreurs disponibles pour les handles sont beaucoup plus nombreux que pour les FCB). Le traitement des erreurs est optimal et donc plus facile.

c) La plupart des futures possibilités des nouvelles versions du DOS pour l'IBM-PC passeront par les handles, et ne seront pas utilisables via les FCB (par exemple le multi-utilisateurs). Pour préserver une possibilité d'extension du programme, il convient donc de traiter les fichiers avec des handles; ils resteront ainsi utilisables au fur et à mesure des versions successives de MS-DOS ou d'OS/2, ou sous un système d'exploitation totalement différent de MS-DOS, pour peu qu'un compilateur C existe pour ce système.

Cependant il faut savoir que les fichiers de type FCB peuvent être gérés en langage C par un ensemble de fonctions plus fourni. Le choix de la méthode dépendra donc essentiellement de l'application à réaliser.

STDIN, STDOUT, ET STDER

Le langage C reconnaît donc trois dispositifs standards d'Entrées/Sorties. Comme par hasard, ces trois fichiers (nous avons vu qu'ils étaient assimilables à des fichiers textes) ont chacun un handle associé : `stdout` est le 0, `stdin` le 1, et `stderr` le 2. D'autre part, les trois dispositifs standard ont également un FCB associé, et peuvent donc être utilisés par les fonctions basées sur les FCB. Les fichiers du programme peuvent utiliser les handles supérieurs à 2.

`stdin` signifie "STandarD INput", soit entrée standard. Représente le canal par lequel l'instruction `scanf()`, par exemple, reçoit les données. Si rien de spécial n'est précisé à l'appel (pas de redirection) ni dans le programme, `stdin` est implicite, et adresse le clavier. Le handle numéro 0 lui est réservé.

`stdout` signifie "STandarD OUTput", soit sortie standard. Représente le canal par lequel l'instruction `printf()`, par exemple, envoie les données. Si rien de spécial n'est précisé à l'appel (pas de redirection) ni dans le programme, `stdout` est implicite, et adresse l'écran. Il utilise le handle numéro 1.

`stderr` signifie "STandarD ERRor", soit erreur standard. Représente le canal sur lequel sont envoyés les messages-erreur lors de l'exécution. `stderr` est généralement branché sur l'écran, comme `stdout`, ou sur un fichier qui reçoit les messages-erreur. Il lui est attribué le handle 2.

Dans toute instruction traitant les entrées, `stdin` est implicite : il est inutile de le préciser, de même que `stdout` est implicite lors des instructions de sorties.

Il est important de se souvenir que `stdin`, `stdout` et `stderr` fonctionnent exactement comme des fichiers textes. Notamment, la réception ou l'envoi d'un caractère `EOF` (End Of File), soit `Ctrl-Z`, indique la fin des émissions ou réceptions. Si `stdout` est par exemple utilisé pour transmettre un `Ctrl-Z`, cela provoquera la fermeture de ce canal. Si `stdout` a été redirigé sur un fichier, celui-ci sera fermé.

Tampon ou non ?

Nous allons parler de tampon dans quelques lignes. Mais il faut que vous reteniez que les fonctions travaillant à base de handle fonctionnent en mode caractères, et n'utilisent pas de tampon. En revanche, les fonctions de fichiers type `FILE` utilisent des tampons. Par exemple, la fonction `printf` n'utilise pas de tampon (sauf celui associé au clavier !). Toutefois, C autorise l'utilisation manuelle de tampons pour les fichiers avec handle, mais le programme doit alors les gérer lui-même.

NOTION DE TAMPON

Vous avez sans doute plusieurs fois entendu parler de tampon au sujet des fichiers ou du clavier. Qu'est-ce qu'un tampon ?

Un première remarque est importante. L'endroit de l'ordinateur où le stockage des informations est le plus pratique à manipuler et le plus rapide à traiter, c'est la mémoire. Comme son nom l'indique, la mémoire vive (ou RAM) est le seul dispositif où l'on puisse stocker une information pour la déplacer ou la modifier de façon dynamique.

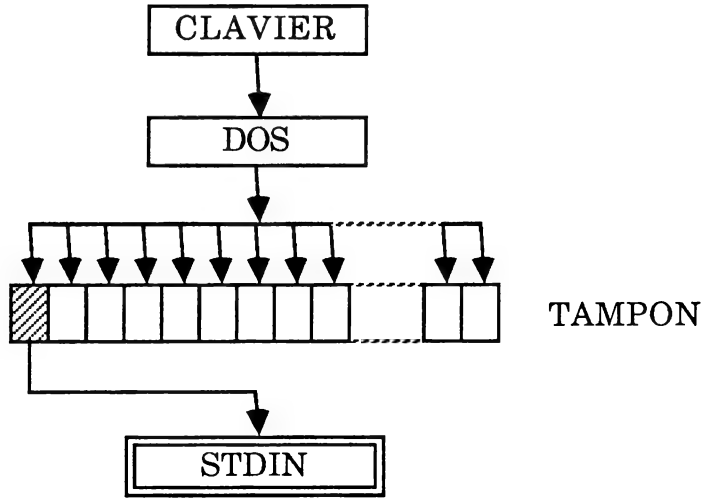
En effet, nous ne pouvons pas écrire sur un clavier. Et les touches n'ont pas de mémoire : si l'utilisateur relâche une touche, rien n'indique plus qu'il l'a enfoncée. Si par malheur nous ne contrôlons pas le clavier à cet instant précis, le programme ne saura jamais que l'utilisateur a appuyé la touche. Nous sommes alors obligés de surveiller constamment le clavier, au besoin en cours de calcul, ce qui est peu pratique.

Le rôle d'un tampon est de supprimer cette obligation.

Un tampon est une zone mémoire (donc facile à modifier et à traiter) qui retient, jusqu'à nouvel ordre, une information, généralement en transit. Lorsque les échanges constants entre le processeur et un dispositif recevant ou transmettant des données ne sont pas possibles, on utilise un tampon.

Pour le clavier, par exemple, le tampon géré par le DOS lui-même est constitué de 16 octets, et peut donc retenir jusqu'à 16 appuis de touches, que le programme peut venir récupérer un à un lorsqu'il en a le temps.

Tampon du DOS



Par analogie, vous devez comprendre qu'un disque n'est guère plus pratique qu'un clavier : on ne peut y stocker ou y lire qu'un seul octet à la fois. Et à chaque fois, il faut positionner la tête de lecture/écriture sur le bon secteur, à la bonne piste, en faisant attention à l'endroit où doit se trouver le fichier ou plus précisément l'octet que l'on désire lire ou écrire.

C'est pourquoi, dans la majorité des langages, on utilise aussi des tampons pour les fichiers. Par exemple, on utilisera un tampon de 2 Ko de mémoire (2048 octets).

Ensuite, la méthode est la suivante : au lieu d'écrire un octet directement sur le disque dans le fichier voulu, on le place dans le tampon qui lui est associé. On écrit ainsi les 2048 premiers octets à une vitesse record. Lorsque le tampon est plein, il faut le vider avant de continuer. La routine qui est chargée d'écrire un octet dans le fichier remarque que le tampon est plein. Elle procède alors, de façon transparente, aux opérations suivantes :

- Se positionner à l'endroit voulu sur le disque.

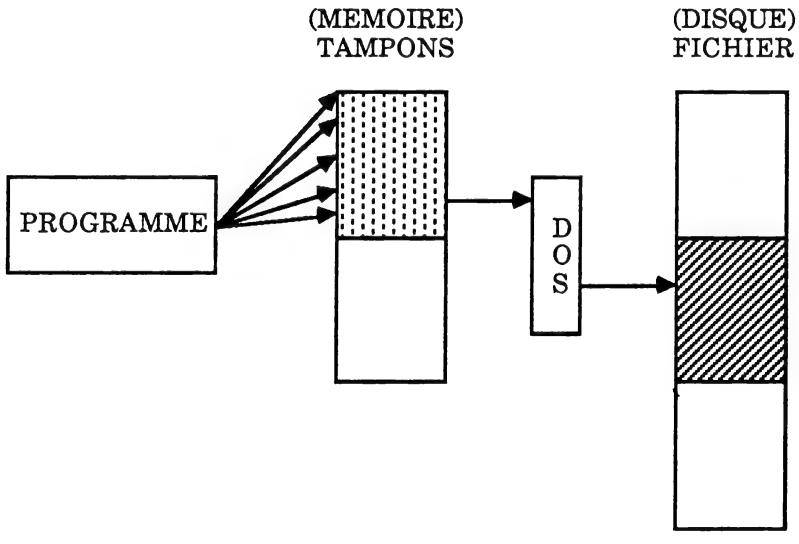
- Ecrire le contenu du tampon.

- Noter que celui-ci est vidé.

- Ecrire, comme si rien ne s'était passé, l'octet suivant dans le tampon.

Bien entendu, l'ensemble des opérations est beaucoup plus rapide, puisqu'on ne positionne la tête du lecteur de disque que tous les 2048 écritures !

Tampon disque



Plus le tampon sera grand, plus les opérations seront rapides. On diminue d'autant les accès aux dispositifs physiques.

Pour la lecture d'un fichier, le principe est le même. Lorsque le programme demande la lecture d'un octet du fichier, la routine de lecture regarde si un octet est disponible dans le tampon. Si oui, elle le fournit et note qu'elle l'a lu (il n'est plus disponible). Si aucun octet n'est plus disponible (les 2048 ont été lus, ou bien c'est la première lecture), la routine s'arrête un instant, le temps de lire 2048 nouveaux octets dans le fichier.

Notez bien que le fonctionnement par tampon n'a pour but que d'accélérer le traitement. Il ne modifie absolument pas la logique des opérations.

Enfin, un tampon étant toujours situé en mémoire, il est évident qu'avant de fermer un fichier en écriture, il faut vider le tampon, sinon on risque de perdre les dernières modifications. Nous reparlerons de ceci lorsque nous examinerons les fonctions `fflush()` et `fclose()`.

Handle sans tampon

Comme nous l'avons indiqué plus haut, les fonctions basées sur les handles n'utilisent pas de tampon. Cela devrait signifier que les E/S standards, notamment, ne sont pas stockées temporairement dans un tampon. Mais bien entendu ces E/S spéciales disposent également d'un accès par FCB, et donc d'un tampon. L'absence de tampon concerne les autres fichiers gérés par handle.

L'inconvénient, c'est qu'il faut alors traiter les informations caractère par caractère, en ayant bien conscience que ceux-ci sont lus ou écrits à chaque fois.

Mais ceci a une explication : en effet, nous avons vu que le principe du handle est de simplifier la gestion, les redirections, et autres manipulations très modernes. Or l'absence de tampon est justement une des raisons pour laquelle ce style de manipulations est aussi souple. En effet, vous pouvez par exemple parfaitement rediriger une écriture en plein milieu du traitement : elle prendra effet dès la prochaine écriture, ce qui serait impossible si le traitement passait par un tampon.

Inversement, C procure des fonctions pour travailler sur des lignes de caractère et non uniquement des caractères. Une sorte de mini-tampon cohérent. On dispose aussi de fonctions capables de traiter un tampon , mais celui-ci doit être géré par le programme.

Revenons sur le programme NUMEROTE.C

Ce programme de numérotation va nous permettre d'expliquer plus simplement tout ceci.

Repassez sous DOS si ce n'est déja fait, et tapez NUMEROTE tout seul. Puis tapez une phrase assez longue et la touche `<enter>`.

Comme vous pouvez le constater, votre phrase est traitée d'un bloc. Or le programme ne comporte qu'une seule instruction de lecture de caractères : `c=getchar()`. Cette fonction a pour effet de lire un caractère sur `stdin`, c'est à dire dans le clavier puisque nous n'avons pas redirigé `stdin` (nous allons voir trés bientôt la fonction `getchar()`).

Mais vous constatez aussi, en lisant le programme que, théoriquement, pour chaque caractère lu, celui-ci est envoyé sur `stdout`, précédé de la valeur de la variable `ligne` si c'est le premier caractère d'une ligne.

Manifestement ce n'est pas le cas, puisque l'écran (`stdout`) ne reçoit les caractères tapés et le numéro de ligne que lorsque vous tapez `<enter>`.

Pourtant, la fonction `getchar()` n'utilise pas de tampon, ni la fonction `putchar()`. Alors pourquoi semble-t-il tout de même y avoir un tampon ?

C'est là que se cache la ruse : les E/S standards (`stdin`, `stdout` et `stderr`) possèdent un tampon, bien qu'elles soient gérées par handle. En clair, `stdin` ne rend les caractères disponibles que lorsque `<enter>` est frappé ou reçu. Ensuite, les caractères précédant ce `<enter>` sont disponibles un à un comme si le tampon n'existait pas aux yeux du programme.

C'est pour cela que le programme les récupère et les écrit un à un , sans utiliser de tampon, comme doit le faire `getchar()`, uniquement lorsque vous tapez `<enter>`; car tant que vous ne l'avez pas fait, aucun caractère n'est disponible sur `stdin` ! Notez que `stdout`, lui, n'est pas ne possède pas de tampon visible non plus.

Maintenant, redirigez `stdin`, par exemple en demandant sous DOS la commande suivante :

```
DIR | NUMEROTE
```

Dans ce cas le `stdin` de NUMEROTE est constitué par les caractères envoyés sur `stdout` par la commande DIR.

Et nous l'avons vu, `stdout` n'a pas de tampon puisque par principe, ce dispositif reçoit un caractère à la fois. Dans ce cas, le traitement est réellement effectué caractère après caractère, et `stdin` n'est plus "tamponné sans en avoir l'air".

Vous remarquez donc que, sans aucune modification du programme et sans que cela perturbe l'effet final, le fonctionnement associé à `stdin` est fondamentalement différent, et pourtant `getchar()` reçoit toujours ses caractères un à un dès que ceux-ci sont disponibles !

C'est là que résident la puissance et la souplesse des E/S standards et par handle.

Et enfin , les tampons sans tampon ...

Maintenant nous devons révéler le scoop final : en langage C, les fonctions de type handle, qui travaillent donc théoriquement sans tampon, permettent quand même d'utiliser des tampons.

Cette particularité permet d'avoir les avantages du tampon avec ceux des handles, ce qui n'est théoriquement pas possible.

Mais attention : les fonctions donnant accès à cette possibilités ne gèrent pas elles-mêmes les tampons. Il appartient au programme appelant de gérer celui-ci. Lors d'une écriture, vous devrez explicitement écrire dans le tampon, puis demander l'écriture du tampon que vous aurez rempli. Mais c'est mieux que rien.

RESUMONS-NOUS

Vous commencez peut-être à avoir l'esprit un peu embrouillé, entre les FCB, les handle, les tampons, les redirections, les E/S standards et les autres. Ne vous inquiétez pas c'est normal. Faisons le point.

a) Il existe trois dispositifs "standards" : `stdin`, `stdout` et `stderr`. Ce sont des dispositifs qui peuvent être redirigés sur demande par le DOS. Ils fonctionnent avec un tampon transparent et par un handle ou un FCB, au choix.

b) Les fichiers fonctionnant par des handles n'ont pas de tampon par le DOS, mais les fonctions de Turbo C s'y référant utilisent une adresse de tampon, qui doit être géré par le programme.

c) Les fichiers fonctionnant par des FCB sont "tamponnés" automatiquement par Turbo C, le programme n'a pas à s'en préoccuper.

d) Les fonctions travaillant sur `stdin`, `stdout` et `stderr` sont "tamponnées" de façon transparente lorsque c'est possible, mais elles fonctionnent dans tous les cas en mode caractère et comme si les dispositifs branchés étaient des fichiers texte sans tampon.

DECLARATION DE FICHIER

Maintenant que nous avons examiné tout ceci d'un point de vue plutôt théorique, il est temps de revenirr au niveau de Turbo C.

Turbo C possède trois types de fonctions travaillant sur les E/S.

- Les fonctions dites "de base" travaillent avec des handles (un nombre de type `int`) avec un pseudo-tampon qui doit être rempli ou lu par le programme (gestion non transparente).

- Les fonctions dites "d'E/S standards" travaillent sur les caractères disponibles ou envoyés sur `stdin`, `stdout` ou `stderr`. Il n'y a pas de tampon, sauf si `stdin` est branché sur le clavier mais dans ce cas la gestion est assurée par

DOS ou les fonctions, entre chaque fin de ligne, de façon transparente. Ces fonctions autorisent le traitement des redirections. Par défaut, elles travaillent avec le clavier et l'écran.

- Les fonctions dites "normales" travaillent avec un bloc de contrôle (FCB) de fichier, auquel correspond un type C nommé `file` Ces fonctions gèrent des tampons de façon transparente.

Bien entendu, il est possible de lire ou écrire sur les dispositifs E/S de ces trois types. Chaque fonction de C n'appartient qu'à un des trois types. Pour les types de base et normaux, il faudra ouvrir et fermer les fichiers. Pour le type E/S standard, comme il n'existe que trois dispositifs et que ceux-ci sont ouverts en permanence, il est inutile de procéder à leur déclaration.

Déclaration d'un type de base.

Un fichier utilisé par les fonctions de base, rappelons le, est tamponné par le programme. La gestion de ce tampon n'est pas automatique.

D'autre part, un tel fichier doit posséder un handle, donc un numéro qui lui est réservé dans la liste de ceux disponibles. Pour cela, il est nécessaire d'ouvrir un tel fichier. Nous verrons très bientôt les fonctions disponibles pour cela.

Déclaration d'un type normal.

Un fichier de type normal doit posséder un FCB. Le tampon est géré automatiquement, mais le compilateur (ou le programme lors de l'exécution) doit bien entendu créer ce FCB. Il est donc aussi nécessaire d'ouvrir (déclarer) un tel fichier. Notez d'autre part que le type `file` est associé à ces fichiers et doit intervenir dans leur déclaration et leur traitement. Nous verrons en détail les fonctions disponibles pour cela.

CHAPITRE 15

LES FICHIERS DE BASE

Les fichiers de base sont gérés par un handle, c'est à dire par un numéro de type `int`.

Notez, c'est important, les deux points suivants :

- Bien que `stdout`, `stdin` et `stderr` soient utilisables par des fonctions qui leurs sont propres, ils sont également accessibles par les fonctions des fichiers de base. Leurs numéros d'accès, ou handle, sont alors respectivement 0, 1 et 2. Ceci permet d'utiliser les possibilités de" tampon manuel".

- Puisque les numéros 0 1 et 2 sont réservés par principe à `stdout`, `stdin` et `stderr`, il est inutile de les ouvrir car l'exécuteur du programme s'en occupe. Ces trois fichiers sont constamment disponibles.

Corollaire de ceci, les handles des autres fichiers ne peuvent commencer qu'à partir du numéro 3.

Créer un fichier de base

creat()

Syntaxe :

```
#include <io.h>
#include <stat.h>
char     *nom_fichier;
int      mode;
int      handle;

handle = creat(nom_fichier,mode);
```

`creat()` est la fonction pour créer un fichier de base ou le remettre à zéro (s'il existait déjà). `mode` indique quel sera l'attribut du fichier (ecture seule, écriture seule, lecture/écriture).

Remarques :

<io.h> contient la déclaration de `creat()`, <stat.h> celle des valeurs possibles pour le paramètre mode.

`nom_fichier` est du type chaîne accessible par un pointeur. On passe une adresse (par exemple le nom de la chaîne).

`mode` correspond à l'un des paramètres suivants :

`S_IWRITE`	: On pourra uniquement écrire dans ce fichier.
`S_IREAD`	: On pourra y lire uniquement.
`S_IWRITE | S_IREAD`	: On pourra y lire ou y écrire.

La valeur renvoyée par la fonction (type `int`) est le numéro de handle attribué. Si la création n'a pas été possible, elle renvoie `-1` et la variable globale `errno` contient le code de l'erreur obtenue. Si <errno.h> est inclus, ce code est l'un des suivants :

`ENOENT`	:	Chemin d'accès ou fichier non trouvé.
`EMFILE`	:	Trop de fichiers ouverts (20 maximum sous TURBO C).
`EACCES`	:	Accès interdit (par exemple multi-utilisateurs).

Exemple :

```
/*-----------------------------------------*/
/* CREAT.C                                 */
/* Exemple de création de fichier type HANDLE */
/* Avec la fonction creat()                */
/*-----------------------------------------*/

#include <io.h>
#include <stat.h>

main()
{
  char  *nom_fichier = "TOTO.FIC";
  int  num_handle;

  puts("Attention, je vais créer le fichier TOTO.FIC");
  puts("en écriture.");
  num_handle = creat(nom_fichier, S_IWRITE);
  if (num_handle == -1)
    puts("Désolé, je n'ai pas pu !");
  else
  {
    puts("Tout va bien, TOTO.FIC existe maintenant.");
  }
  exit(); /* sortir en fermant tous les fichiers */
}
```

Ouvrir un fichier de base existant

open()

Syntaxe :

```
#include <io.h>
#include <stat.h>
#include <fcntl.h>
char     *nom_fichier;
int      handle;
handle = open(nom_fichier,acces);
handle = open(nom_fichier,O_CREAT,mode);
acces:
         O_RDONLY
         O_WRONLY
         O_RDWR
         O_APPEND
         O_CREAT
         O_TRUNC
         O_BINARY
         O_TEXT
mode  :
         S_IREAD
         S_IWRITE
         S_IREAD | S_IWRITE
```

La fonction `open()` ouvre un fichier créé auparavant. Si l'accès est `O_CREAT`, il faut également préciser le mode de création (attribut du fichier). On peut combiner plusieurs valeurs de l'accès en les séparant par un `|` (OU).

Remarques :

`<io.h>` contient la déclaration de `open()`, `<stat.h>` et `<fcntl.h>` celle des valeurs possibles des paramètres `mode` et `acces`.

`nom_fichier` est du type chaîne accessible par un pointeur. On passe une adresse (par exemple le nom de la chaîne).

`acces` est de type `int`, représenté par l'une des constantes suivantes (définies dans `<stat.h>`) :

- `O_RDONLY` : Lecture seule.
- `O_WRONL` : Ecriture seule.
- `O_RDWR` : Lecture et écriture.
- `O_APPEND` : Les écritures seront systématiquement ajoutées en bout de fichier.
- `O_CREAT` : sans effet s'il existe déja, sinon le fichier est créé selon le mode attribué (voir ci-dessous).
- `O_TRUNC` : Le fichier est vidé, mais son attribut ne change pas (lecture ou écriture seule, etc...)

`O_BINARY` : MODE BINAIRE (le caractère `Ctrl-Z` n'indique pas la fin de fichier et `Ctrl-M` pas une fin de ligne).
`O_TEXT` : MODE TEXTE (l'inverse du précédent !).

Pour combiner plusieurs de ces modes, les séparer par un OU ("|") (voir l'exemple plus bas).

`mode` est indiqué si l'accès est `O_CREAT`, et peut recevoir les mêmes valeurs que pour la fonction `creat()` :

`S_IREAD` : Lecture seule.
`S_IWRITE` : Ecriture seule.
`S_IREAD | S_IWRITE` : Lecture ET écriture.

La valeur `int` renvoyée par la fonction est le numéro de handle attribué. Si l'ouverture n'a pas été possible, elle renvoie `-1` et la variable globale `errno` contient le code de l'erreur obtenue. Si `<errno.h>` est inclus, ce code est l'un des suivants :

`ENOENT` : Chemin d'accès ou fichier non trouvé.
`EMFILE` : Trop de fichiers ouverts (20 maximum sous TURBO C).
`EACCES` : Accès interdit. (par exemple multi-utilisateurs).
`EINVACC`: Code `acces` inconnu (voir les codes autorisés ci-dessus).

Exemple :

```
/*-----------------------------------------------*/
/* OPEN.C                                        */
/* Exemple d'ouverture de fichier type HANDLE    */
/* Avec la fonction open()                       */
/* EXECUTER APRES CREAT.C, SVP !!!!              */
/*-----------------------------------------------*/
#include <io.h>
#include <fcntl.h>

main()
{
   char  *nom_fichier = "TOTO.FIC";
   int   num_handle;

   puts("Attention, je vais OUVRIR le fichier TOTO.FIC");
   puts("en lecture/écriture et mode binaire !");
   num_handle = open(nom_fichier,O_RDWR | O_BINARY);
   if (num_handle == -1)
        puts("Désolé, je n'ai pas pu !");
   else
   {
        puts("Tout va bien, TOTO.FIC est ouvert.");
   }
   exit(); /* sortir en fermant tous les fichiers */
}
```

Fermer un fichier de base

close()

Syntaxe :

```
#include <errno.h>
#include <io.h>
int      handle;
int      echec;
echec = close(handle);
```

La fonction `close()` ferme un fichier. Celui-ci ayant précédemment été ouvert ou créé par `creat()` ou `open()`, il lui est attribué un handle. C'est ce numéro, le handle, qu'il faut envoyer à la fonction `close()`.

Remarques :

Si la fonction s'est correctement effectuée, la variable `echec` prend la valeur `0` (faux), sinon elle prend la valeur `-1` (vrai). Si le handle spécifié n'est pas attribué, il y a erreur, et la variable globale `errno` prend la valeur `EBADF` (Bad File Number, soit mauvais numéro de fichier).

Exemple :

```
/*-------------------------------------------*/
/* CLOSE.C                                   */
/* Exemple de fermeture de fichier type HANDLE */
/* Avec la fonction creat()                  */
/*-------------------------------------------*/

#include <errno.h>
#include <io.h>
#include <stat.h>

main()
{
   char  *nom_fichier = "TOTO.FIC";
   int   num_handle;

   puts("Attention, je vais créer le fichier TOTO.FIC");
   num_handle = creat(nom_fichier, S_IWRITE | S_IREAD);
   if (num_handle == -1)
        puts("Désolé, je n'ai pas pu !");
   else
   puts("Tout va bien, TOTO.FIC existe maintenant.");
```

```
    /* sortir en fermant le fichier */
    if (close(num_handle))
    {
         puts("Impossible de fermer le fichier !");
         printf("Erreur No %hd\n",errno);
    }
    else
    {
         puts("Le fichier est bien fermé.");
    }
}
```

Ecrire des données dans un fichier de base

write()

Syntaxe :

```
#include <errno.h>
#include <io.h>
int      handle;
XXX      *tampon;
int      nb_octets;
int      echec;

echec = write(handle,tampon,nb_octets);
```

La fonction `write()` écrit le contenu d'un tampon rempli par le programme dans le fichier indiqué. L'appel doit également indiquer le nombre d'octets qui seront pris dans le tampon. Le type `XXX` dans les déclarations ci-dessus représente le type du tampon. Il est quelconque, c'est le programme qui doit le gérer. `write()` écrit un retour chariot et un passage à la ligne pour chaque retour chariot envoyé.

Remarques :

Si l'écriture s'est correctement effectuée, elle renvoie `0` (faux), sinon elle renvoie `-1` (vrai) et la variable `errno` contient l'un des codes suivants :

`EACCES` : Accès refusé
`EBADF` : Handle non attribué, impossible d'écrire.

Si le fichier a été ouvert par `open()` en mode `O_APPEND`, `write()` écrit les données systématiquement au bout du fichier.

On peut écrire avec `write()` directement sur `stdout` et `stderr` sans ouvrir ces fichiers : il suffit d'utiliser un numéro handle ayant les valeurs `0` et `2` respectivement pour `stdout` et `stderr` (voir exemple `WRITE2.C` ci-dessous).

Exemple 1:

```
/*-----------------------------------------*/
/* WRITE1.C                                */
/* Exemple d'écriture dans un fichier HANDLE */
/* Avec la fonction write()                */
/*-----------------------------------------*/

#include <errno.h>
#include <io.h>
#include <stat.h>
#include <string.h>

main()
{
   char  *nom_fichier = "TOTO.FIC";
   int   num_handle;
   char  *phrase = "Essai d'écriture";
   int   nb_octets;
   int   resultat;
   puts("Attention, je vais créer le fichier TOTO.FIC");
   puts("en écriture.");
   num_handle = creat(nom_fichier, S_IWRITE);
   if (num_handle == -1)
        puts("Désolé, je n'ai pas pu !");
   else
   {
        puts("Tout va bien, TOTO.FIC existe maintenant.");
   }
   nb_octets = strlen(phrase);
   resultat = write(num_handle , phrase , nb_octets);
   if (resultat == -1)
        puts("Erreur, impossible d'écrire dans le fichier !");
   else
   {
        puts("Pas de problème, la phrase a été écrite.");
        puts("Essayez TYPE TOTO.FIC");
   }
   if (close(num_handle))
   {
        puts("Impossible de fermer le fichier !");
        printf("Erreur No %hd\n",errno);
   }
   else
   {
        puts("Le fichier est bien fermé.");
   }
   exit();  /* sortir en fermant tous les fichiers */
}
```

Exemple 2:

```
/*-----------------------------------------*/
/* WRITE2.C                                */
/* Exemple d'écriture sur la sortie standard */
/* Avec la fonction write()                */
/*-----------------------------------------*/

#include <stdio.h>
#define STDOUT 0
#define STDIN  1
#define STDERR 2

main()
{
   char  *phrase = "Bonjour tout le monde !\n\n";
   int   nb_octets;
   int   resultat;

   puts("Ecriture directe sur stdout avec write() :\n\n");
   nb_octets = strlen(phrase);
   resultat = write( STDOUT , phrase , nb_octets);
   if (resultat == -1)
   {
        write(STDERR,"Erreur, impossible d'écrire sur stdout",38);
        write(STDERR,"Donc ce message est envoyé par stderr !",39);
   }
   else
   {
        printf("\n%d octets ecrits.\n\n\n",resultat);
   }
}
```

Lire des données dans un fichier de base

read()

Syntaxe :

```
#include <errno.h>
#include <io.h>
int     handle;
XXX     *tampon;
int     nb_octets;
int     echec;

echec = read(handle,tampon,nb_octets);
```

La fonctions `read()` lit le contenu d'un fichier dans un tampon. L'appel doit également indiquer le nombre d'octets qui seront pris dans le tampon. Le type

`xxx` dans les déclarations ci-dessus représente le type du tampon. Il est quelconque, c'est le programme qui doit le gérer.

Remarques :

-Si la lecture s'est correctement effectuée, elle renvoie le nombre d'octets stockés dans le tampon. Si une erreur est survenue, `-1` est renvoyé, et si la fin du fichier est atteinte, la fonction renvoie `0`.

-Si le fichier a été ouvert en mode écriture (par `creat()` ou `open()` en mode `O_CREAT` ou `O_APPEND`, par exemple), une lecture renverra toujours `0` (puisque le fichier est supposé vide).

Exemple :

```
/*-----------------------------------------*/
/* READ1.C                                 */
/* Exemple de lecture dans un fichier HANDLE */
/* Avec la fonction read()                 */
/* Executer le programme WRITE1.C d'abord  */
/*-----------------------------------------*/

#include <errno.h>
#include <fcntl.h>
#include <stddef.h>
#include <alloc.h>

main()
{
  char  *nom_fichier = "TOTO.FIC";
  int  num_handle;
  char  *phrase;
  int  nb_octets;

  puts("J'ouvre le fichier TOTO.FIC en lecture.");
  num_handle = open(nom_fichier,O_RDONLY);

  if (num_handle == -1)
  {
    puts("Désolé, je n'ai pas pu !");
    exit();
  }

  if ((phrase = (char *) malloc(80)) == NULL)
  {
    puts("Impossible de créer le tampon.");
    exit();
  }
  nb_octets = read(num_handle , phrase , 80);
```

```
  if (nb_octets == -1)
    puts("Erreur, impossible d'écrire dans le fichier !");
  else if (nb_octets == 0)
    puts("Le fichier est vide");
  else
  {
    puts("Ce que j'ai lu dans le fichier :\n\n");
    puts(phrase);
    printf("\n\nJ'ai lu %d octets.\n",nb_octets);
  }

  if (close(num_handle))
  {
    puts("Impossible de fermer le fichier !");
    printf("Erreur No %hd\n",errno);
  }
  else
  {
    puts("Le fichier est bien fermé.");
  }
  /* sortir en fermant tous les fichiers */
  exit();
}
```

On peut lire avec `read()` directement sans ouvrir ce fichier; il suffit d'utilser le handle 1. Exemple :

```
/*-----------------------------------------*/
/* READ2.C                                 */
/* Exemple de lecture sur l'entrée standard  * /
/* Avec la fonction read()                  */
/*-----------------------------------------*/

#include <stdio.h>
#define STDOUT 0
#define STDIN  1
#define STDERR 2

main()
{
  char  phrase[80]; /* Un tableau pour changer un peu !*/
  int  nb_octets = 80;
  int  resultat;

  puts("Lecture directe sur stdin avec read() :\n\n");

  resultat = read( STDIN , phrase , nb_octets);
```

```
  if (resultat == -1)
  {
   puts("Impossible de lire sur STDIN !!!");
   exit();
  }
  else
  {
   printf("\n%d octets lus.\n\n\n",resultat);
   puts("J'ai reçu la chose suivante :\n\n");
   write(STDOUT,phrase,resultat);
  }
}
```

Positionnement dans un fichier de base

lseek()

Syntaxe :

```
#include <io.h>
#include <errno.h>
#include <stdio.h>
int      handle;
long     deplacement;
long     resultat;

resultat = lseek(handle , deplacement , position);
position :
         SEEK_SET
         SEEK_CUR
         SEEK_END
```

Permet de débuter la lecture ou l'écriture dans un fichier à un emplacement donné. `position` prend l'une des trois valeurs ci-dessus, représentant respectivement le début du fichier, la posisiont actuelle dans le fichier, et la fin du fichier. Le déplacement est compté en nombre d'octets à partir de cette position.

Remarques :

-Le nombre d'octets de `deplacement` par rapport à `position` est indiqué sous la forme d'un entier 32 bits (type `long` signé). Il peut être négatif.

-La fonction renvoie un entier `long` également, qui prend la valeur `-1L` si une erreur a eu lieu. On récupère alors le code erreur dans la variable globale `errno` si `<errno.h>` est inclus, celle-ci pouvant prendre les valeurs suivantes :

`EBADF` : Handle non attribué.
`EINVAL` : Argument invalide (déplacement hors fichier...)

Exemple :

```
/*-------------------------------------------*/
/* LSEEK.C                                   */
/* Exemple positionnement dans un fichier    */
/* Avec la fonction lseek()                  */
/* EXECUTER APRES CREAT.C, SVP !!!!          */
/*-------------------------------------------*/

#include <errno.h>
#include <io.h>
#include <stdio.h>
#include <fcntl.h>

main()
{
   char  *nom_fichier = "TOTO.FIC";
   int   handle;
   long  deplace;
   long  lresultat;
   int   resultat;
   char  tampon[80];
   int   nb_octets;
   int   i ;

   puts("Attention, je vais OUVRIR le fichier TOTO.FIC");
   puts("en lecture et mode binaire !");
   handle = open(nom_fichier,O_RDONLY | O_BINARY);
   if (handle == -1)
   {
        puts("Désolé, je n'ai pas pu !");
        exit();
   }

   deplace = 10L;
   lresultat = lseek( handle, deplace, SEEK_SET);
   if (lresultat == -1L)
   {
        puts("Impossible de trouver le 10e octet de TOTO.FIC !");
        switch (errno)
        {
        case EBADF :
                printf("Handle %d non attribué\n",handle);
                break;
        case EINVAL :
                printf("Déplacement impossible ou incorrect
                          %ld\n",deplace);
                break;
```

```
        default :
                printf("Erreur inconnue %d\n",errno);
        }
        exit();
    }

    nb_octets = 3;
    resultat = read( handle, tampon, nb_octets);
    if (resultat == -1)
    {
        puts("Impossible de lire les octets 10, 11 et 12");
        switch (errno)
        {
        case EACCES :
                puts("Accès interdit");
                break;
        case EBADF :
                puts("Handle non attribué");
                break;
        default :
                puts("Erreur inconnue.");
        }
        exit();
    }

    puts("Voici les 3 octets lus  (position 10, 11 et 12) :");
    for (i=0 ;i<nb_octets; i++)
    {
        printf("%d : ",deplace+i);
        putchar( *(tampon+i) ); /* Accès identique à tampon[i] */
        puts("");
    }
    exit();
}
```

Conclusion

La manipulation des fichiers par handle est intéressante car elle permet une indépendance des fonctions et du contenu du fichier. En réalité, seul le programme est responsable de ce qu'il place dans le fichier, et de sa récupération.

On obtient donc des manipulations de fichiers extrêmement souples.

La contrepartie de cette souplesse est bien évidemment l'obligation pour le programme de bien gérer ses tampons et son fichier.

CHAPITRE 16

LES FICHIERS STANDARD

Nous avons annoncé au chapitre 14 que les trois fichiers standard d'entrée, de sortie et de message-erreur disposaient d'un statut spécial au sein du langage C.

En effet, non seulement ils sont en permanence ouverts (nous l'avons constaté dans l'exemple de la fonction `write()`, par exemple) mais en plus ils disposent de quelques instructions qui leur sont propres.

L'avantage de ces fonctions est d'assimiler ces trois dispositifs à des fichiers texte déjà ouverts, justement, de sorte que la redirection des entrées et des sorties vers de tels fichiers est très simple.

En revanche, il doit être clair que ces fonctions ne peuvent absolument pas être utilisées sur d'autres dispositifs de façon fiable. Elles constituent une restriction des fonctions que nous examinerons dans la partie suivante de ce chapitre, et qui, elles, adressent n'importe quel type de fichier.

Notez que les fonctions sur les fichiers standard adressent implicitement le handle qui leur est attribué : 0 pour `stdout`, 1 pour `stdin` et 2 pour `stderr`.

Toutefois, toutes les fonctions que nous examinerons dans la partie suivante (fichiers normaux) peuvent être également être utilisées sur ces 3 dispositifs, grâce aux déclarations de `<stdio.h>` qui incluent un FCB pour `stdin`, `stdout` et `stderr`. Un exemple de cette possibilité est constitué dans ce qui suit par la fonction `ungetc()`, qui n'appartient pas aux fonctions sur les fichiers standard, mais qui est utilisée ici pour replacer un caractère dans `stdin`.

Rappelons enfin que, sauf redirection à partir du DOS, `stdin` est connecté au clavier (lequel est associé à un tampon par le système), `stdout` à l'écran, et `stderr` également à l'écran.

Lire un caractère sur stdin

getchar()

Syntaxe :

```
#include <stdio.h>
int     caract;

caract = getchar();
```

La fonction `getchar()` lit tout simplement un caractère dans le tampon de `stdin`. Si une fin de fichier est obtenue (`Ctrl-Z` au clavier, par exemple, suivi d'un appui sur `<enter>` pour rendre le tampon disponible), la valeur `-1` est

renvoyée. Elle est représentée par la constante `EOF` (End Of File, soit fin de fichier).

Remarques :

Notez que la fonction ne renvoie pas un caractère mais un entier de type `int`. Il est donc possible de l'ajouter dans un tableau de caractère.

Remarquez également que le clavier étant géré par un tampon, les caractères tapés ne sont disponibles par `getchar()` que lorsque la touche `<enter>` a été frappée. Ils sont à partir de ce moment disponibles un à un par la fonction, comme s'ils avaient été tapés et récupérés dynamiquement.

Notez enfin que `getchar` attend qu'un caractère soit disponible.

Exemple :

```
/*-------------------------------------------------------*/
/* GETCHAR.C                                             */
/* Exemple de lecture dans stdin par getchar()           */
/* Ce programme retourne une chaîne ou plusieurs.        */
/* Essayez les exemples suivants:                        */
/*                                                       */
/* DIR | GETCHAR                                         */
/* DIR | GETCHAR | GETCHAR                               */
/*                                                       */
/* Notez que l'intégralité du texte disponible sur stdin */
/* est chargée avant le retournement (dans la limite des */
/* 30000 octets réservés ce qui fait que le texte est    */
/* retourné de droite à gauche ET de bas en haut.        */
/*-------------------------------------------------------*/

#include <stdio.h>

#define MAXI 30000

main()
{
   int   lu;
   char  tampon[MAXI];
   short limite = 0, indice = 0;

   puts("Tapez une phrase finie par ENTER, puis");
   puts("un Ctrl-Z seul sur la ligne suivi de ENTER\n\n");

   /* Récupérer la phrase un caractère à la fois sur     */
   /* stdin une fois ENTER frappé. S'arrêter à \0, qui   */
   /* indique la fin de ligne.                           */

   while( ( (tampon[limite]=getchar()) != EOF ) && (limite<MAXI) )
         limite++;
```

```
   for (indice = --limite; indice>=0 ; indice--)
   printf("%c",tampon[indice]);
}
```

Ecrire un caractère sur stdout

putchar()

Syntaxe :

```
#include <stdio.h>
char    caract;

putchar(caract);
```

Ecrit un caractère dans le tampon de `stdout`.

Ecrire un caractère sur stderr

puterr()

Syntaxe :

```
#include <stdio.h>
char    caract;

puterr(caract);
```

Ecrit un caractère dans le tampon de `stderr`.

Remettre un caractère lu dans stdin

ungetc()

Syntaxe :

```
#include <stdio.h>
int     resultat;
char    caract;

resultat = ungetc(caract,stdin);
```

Remet en place un caractère dans le tampon de `stdin`. Ce caractère sera le prochain lu par `getchar()`. La fonction renvoie `EOF` (`-1`) si l'on se trouvait sur la fin du fichier (caractère `Ctrl-Z`), dans ce cas le caractère n'est pas replacé dans le tampon.

Exemple :

Pour tester si Ctrl-Z est disponible sur stdin sans perdre le prochain caractère, il suffit d'insérer une ligne comme :

```
if (ungetc(getchar()) == EOF)
   {
        .. C'est la fin du fichier stdin ! ...
   }
```

Lire une chaîne sur stdin

gets()

Syntaxe :

```
#include <stdio.h>
#include <stddef.h>
char     *tampon;
char     *resultat;

resultat = gets(tampon);
```

La fonction gets() lit une chaîne (terminée par un retour-chariot, ou une fin de fichier Ctrl-Z / EOF) dans stdin, et la charge dans le tampon dont l'adresse est passée en paramètre. Si une erreur est survenue, l'adresse renvoyée est un pointeur NULL.

Remarques :

- Le tampon doit avoir été réservé, puisqu'on ne connaît pas la longueur de la chaîne à priori (en tout cas, la fonction ne peut pas savoir combien d'octets ont été réservés pour son stockage).

- Pour tester si le pointeur renvoyé est NULL, il faut inclure le fichier en-tête <stddef.h>. Si l'opération se déroule normalement, le pointeur renvoyé est identique à celui fourni en paramètre.

Exemple :

```
/*-------------------------------------------------------*/
/* GETS.C                                                */
/* Exemple de lecture dans stdin par gets                */
/* Ce programme retourne une chaîne ou plusieurs.        */
/* Essayez les exemples suivants:                        */
/*                                                       */
/* DIR | GETS                                            */
/*                                                       */
/* Si vous avez compilé le programme GETCHAR.C :         */
/*                                                       */
/* DIR | GETS | GETCHAR                                  */
/*                                                       */
```

```
/* Notez que le texte est retourné, cette fois-ci, ligne */
/* par ligne et non en entier.                           */
/* Notez aussi qu'il est nécessaire de replacer les      */
/* Passages à la ligne car gets() ne les restitue pas    */
/* dans le tampon de ligne.                              */
/*-------------------------------------------------------*/

#include <stdio.h>
#include <stddef.h>
#include <alloc.h>

#define MAXI 512

main()
{
  int lu;
  char *tampon;
  char *resultat;
  short i ;

  tampon = (char *) malloc(MAXI);
  if (tampon == NULL)
  {
    puts("IMPOSSIBLE de réserver le tampon de ligne");
    exit();
  }

  /* Récupérer les phrases sur stdin ligne par ligne entre */
  /*  chaque appui ENTER ou retour-chariot.          * /

  while ( (resultat = gets(tampon)) != NULL)
  {
    for (i=strlen(tampon); i>=0; i--)
      printf("%c",tampon[i]);
    /* Le caractère d'interligne n'est pas récupéré */
    /* il faut le restituer                 * /
    printf("\n");
  }
}
```

Ecrire une chaîne sur stdout

puts()

Syntaxe :

```
#include <stdio.h>
char    *phrase;
int     resultat;

resultat = puts(phrase);
```

Envoie la chaîne pointée par le paramètre sur `stdout`. La fonction renvoie `EOF` (`-1`) si la fin de fichier est rencontrée ou si une erreur survient. Le passage à la ligne est automatiquement ajouté (`\n`) à la suite de la chaîne.

Lire et interpréter des données sur stdin

scanf()

Syntaxe :

```
#include <stdio.h>
char    *format;
XXX     varX;
YYY     varY;
...
int     resultat;

resultat = scanf(format, &varX, &varY, ...);
```

Lit les données au format indiqué et les stocke dans les zones mémoires pointées. Il doit y avoir correpondance totale entre la chaîne `format` (qui représente, comme dans un format `printf`, les données à recevoir) et les types des variables dont on passe le pointeur. La fonction renvoie soit le nombre de données transmises dans des variables (qui doit correspondre au nombre de pointeurs passés derrière le format), soit `-1` si une erreur est survenue.

`format` est une chaîne (entre guillemets par exemple) contenant des attributs de données (un attribut par variable demandée) et des séparateurs pour ces attributs.

Exemples de formats :

`"%d , %f"` : attendra un entier, une virgule puis un réel.
`"%c%c%c\n"` : attendra trois caractères puis un retour-chariot.

Principaux attributs valides (N représente un nombre entier) :

%d : décimal int.
%Nd : décimal int avec N chiffres.
%hd : décimal short.
%Nhd : décimal short avec N chiffres.
%ld : décimal long.
%Nld : décimal long avec N chiffres.
%f : réel float.
%Nf : réel float avec N chiffres.
%lf : réel double.
%Nlf : réel double avec N chiffres.
%c : un caractère char.
%Nc : N caractères char.
%s : une chaîne terminée par des blancs/tabulations/<enter>.
[^...] : "..." représente les caractères valides pour ce paramètre.
[...] : "..." représente l'ensemble des délimiteurs de ce champ.

Le manuel de référence Turbo C comporte une description détaillée des autres possibilités de la chaîne de format pour scanf. Ceci n'est qu'une petite liste de ces possibilités (enfin, pas si petite que ça tout de même...)

Exemples :

```
/*-----------------------------------------------------------*/
/* SCANF.C                                                   */
/*                                                           */
/* Application de scanf()                                    */
/* Pour tester ce programme :                                */
/*                                                           */
/* C> SCANF <enter>                                          */
/*                                                           */
/* suivi de :                                                */
/*                                                           */
/*    NOM <enter> prenom <enter> age <enter>                 */
/* ou NOM prenom <enter> age <enter>                         */
/* ou NOM prenom age <ENTER>                                 */
/* etc...                                                    */
/*                                                           */
/* ou bien taper :                                           */
/*                                                           */
/* C> COPY CON: TEST.TXT                                     */
/* NOM <enter> prenom <enter> age <enter> Ctrl-Z <Enter>     */
/*                                                           */
/* suivi de :                                                */
/*                                                           */
/* SCANF <TEST.TXT                                           */
/*                                                           */
/*-----------------------------------------------------------*/
```

```
#include <stdio.h>
#include <errno.h>
#include <string.h>

main()
{
  char *nom;
  int age;
  char *prenom;
  char prefere;
  char *resultat;

  /* Réserver la place pour nom et prenom */
  if ((nom = (char *) malloc(40)) == NULL)
  {
    puts("Impossible d'allouer de la mémoire...");
    exit();
  }
  if ((prenom = (char *) malloc(40)) == NULL)
  {
    puts("Impossible d'allouer de la mémoire...");
    exit();
  }

  puts("Bonjour !");
  puts("\n\nQuel est votre nom (en majuscules svp) ?");

  /* Attendre des majuscules ou ' ou un espace : */
  if (scanf("%[ABCDEFGHIJKLMNOPQRSTUVWXYZ' ]",nom) == NULL)
  {
    puts("Ah, je n'ai pas bien compris votre nom.");
    exit();
  }
  puts("Et votre prénom ?");

  /* Attendre une chaîne */
  if (scanf("%s",prenom) == NULL)
  {
    puts("Je suis désolé, je n'ai pas saisi votre prénom...");
    exit();
  }

  /* Attendre un entier */
  puts("Serait-il indiscret de vous demander votre âge ?");
  if (scanf("%d",age) == NULL)
  {
    puts("Je n'ai pas compris.");
    exit();
  }
```

```
  printf("Je suis heureux de vous rencontrer, %s %s.\n",prenom,nom);

  /* Afficher un commentaire suivant l'âge */
  /* Avec Turbo C, une chaîne peut être coupée en plusieurs lignes */
  /* dans le source si rien ne sépare les guillemets qui découpent */
  /* les parties de la chaîne. */
  /* On utilise ici la forme EXP ? EXP1 : EXP2 qui donne la */
  /* valeur EXP1 si EXP est vraie et EXP2 sinon. */

  printf("%s\n",(age < 18) ?
                  "Vous n'avez pas vu Armstrong poser le pied"
                  " sur la Lune. Dommage.\n"
                : "Vous avez vu les Beatles se séparer,"
                " ca n'arrive pas tous les jours !\n");
}
```

Ecrire des données sur stdout

printf()

Syntaxe :

```
#include <stdio.h>
char     *format;
XXX      varX;
YYY      varY;
...
int      resultat;

resultat = printf(format, varX, varY, ...);
```

Prend les données dans les variables et les envoie sur `stdout` au format indiqué. Il doit y avoir correpondance totale entre la chaîne `format` et les types des variables. La fonction renvoie le nombre d'octets finalement envoyés sur `stdout`, ou `EOF` (`-1`) si une erreur est survenue.

`format` est une chaîne (entre guillemets par exemple) contenant des attributs de données (un attribut par variable demandée) et des séparateurs pour ces attributs. Tout ce qui n'est pas un attribut dans la chaîne `format` est envoyé tel quel, et ne correspond pas à une variable de la liste.

Exemples de formats :

- `"%d , %f"` : un entier, une virgule puis un réel.
- `"%c%c%c\n"` : trois caractères puis un retour-chariot.
- `"%P -> %s"` : un pointeur , une flèche et une chaîne.

Principaux attributs valides (`Y` et `Z` représentent des nombres entiers) :

`%d`	: décimal `int`.
`%Y.Zd`	: décimal `int`, minimum `Z` chiffres, maximum `Y` chiffres.
`%hd`	: décimal `short`.
`%Y.Zhd`	: décimal `short` minimum `Z` chiffres, maximum `Y` chiffres.
`%ld`	: décimal `long`.
`%Y.Zld`	: décimal `long` minimum `Z` chiffres, maximum `Y` chiffres.
`%f`	: réel `float`.
`%Y.Zf`	: réel `float` `Z` décimales, `Y` chiffres avant la virgule.
`%lf`	: réel `double`.
`%Y.Zlf`	: réel `double` `Z` décimales, `Y` chiffres avant la virgule.
`%c`	: un caractère `char`.
`%Yc`	: `Y` caractères.
`%s`	: une chaîne terminée par des blancs/tabulations/<enter>.
`%FP`	: pointeur Far, sous la forme XXXX:YYYY (segment:offset)
`%NP`	: pointeur Near sous la forme YYYY (offset)

Le manuel de référence Turbo C comporte une description détaillée des autres possibilités de la chaîne de format pour `printf`. Ceci n'est qu'une petite liste de ces possibilités.

CHAPITRE 17

FICHIERS AVEC FCB

Les fichiers associés à un FCB peuvent être gérés grâce à de trés nombreuses fonctions. Il faut se souvenir que ces fichiers sont associés à une structure `FILE` déclarée dans une en-tête de Turbo C. De plus, comme nous l'avons expressément souligné, les fichiers gérés par FCB sont associés à un tampon, qui est lui-même géré par DOS et les fonctions C de manière totalement transparente.

Toutefois, il faudra utiliser la fonction `fflush()` (voir ci-dessous) de temps en temps, et surtout avant de quitter le programme ou fermer un fichier, sinon le contenu de l'éventuel tampon d'écriture risque d'être perdu sans avoir été physiquement copié sur le disque.

Rappelons enfin que `stdin`, `stdout` et `stderr` peuvent être utilisés directement les fonctions qui suivent. C'est Turbo C qui se charge de leur associer la structure `FILE` appropriée et de les ouvrir dés le lancement d'un programme.

La structure `FILE`, les constantes `stdin`, `stdout` et `stderr`, sont déclarées dans le fichier en-tête `<stdio.h>`, ainsi que les fonctions décrites ci-dessous. Il faudra donc systématiquement inclure ce fichier au début des programmes.

Nous ne passerons pas en revue toutes les fonctions sur les fichiers, mais uniquement les plus importantes. Pour en savoir plus, veuillez vous reporter aux manuels Borland.

Créer un fichier avec FCB.

fopen()

Syntaxe :

```
#include <stdio.h>
#include <stddef.h>
#include <errno.h>
FILE     *fcb1;
char     *nom_fichier;
char     *mode;

fcb1 = fopen(nom_fichier,mode);
```

Ouvre le fichier dont le nom est indiqué, dans le mode précisé. La fonction crée le FCB du fichier et renvoie le pointeur vers la zone qui le contient (ci-dessus, `fcb1`). `mode` est l'une des chaînes suivantes :

`"r"`	(Read)	:	le fichier est ouvert uniquement en mode lecture sur le premier caractère du fichier.
`"w"`	(Write)	:	le fichier est créé et prêt à recevoir des écritures. Il est écrasé (longueur ramenée à 0) s'il existait déja.
`"a"`	(Append)	:	le fichier est créé s'il n'existait pas. Sinon, le pointeur se positionne en mode écriture uniquement, mais au bout du fichier (le contenu déjà stocké n'est pas écrasé).
`"r+"`	(Read+)	:	le fichier est ouvert en lecture/écriture, à son début.
`"w+"`	(Write+)	:	le fichier est recréé (comme `"w"`) mais il est en mode lecture/écriture (il est possible de lire après avoir écrit).
`"a+"`	(Append+)	:	le fichier est ouvert en lecture/écriture et positionné à son bout.

Remarques :

Si le fichier ne peut pas être ouvert, la fonction renvoie le pointeur `NULL` et place le code erreur dans la variable globale `errno`, si le fichier `<errno.h>` est inclus.

La mémoire nécessaire pour stocker le FCB du fichier ouvert est réservée par la fonction `fopen()` elle-même, il est donc inutile de procéder à une allocation pour `fcb1`. Elle ne sera libérée que par `fclose()` (voir cette fonction plus loin).

Exemple :

```
/*-------------------------------------------*/
/* FOPEN.C                                   */
/*                                           */
/* Exemple de création de fichier par fopen() */
/*                                           */
/* Ce programme créée le fichier TOTO.FIC    */
/*                                           */
/*-------------------------------------------*/

#include <stdio.h>
#include <stddef.h>
#include <errno.h>

main()
{
  FILE *controle_bloc;
  char *nom_fichier = "TOTO.FIC";
  char *mode = "w";
```

```
  if ((controle_bloc = fopen(nom_fichier,mode)) == NULL)
  {
    puts("Impossible de créer le fichier !");
    exit();
  }
  puts("Le fichier TOTO.FIC a été créé avec son FCB.");
}
```

Ecrire dans un fichier avec FCB.

fwrite()

Syntaxe :

```
#include  <stdio.h>
#include  <errno.h>
char      *tampon;
FILE      *fcb;
int       longueur;
int       nombre;
int       resultat;

resultat = fwrite(tampon, longueur, nombre, fcb);
```

Ecrit le contenu du tampon dans le fichier dont `fcb` est fourni. Les paramètres `longueur` et `nombre` indiquent la longueur (en octets) et le nombre d'éléments qui se trouvent dans le `tampon`. Ce `tampon` peut être une simple variable, si l'on désire n'écrire qu'un seul élément. La fonction renvoie le nombre d'éléments écrits, ou `-1` si une erreur a eu lieu (le code d'erreur est alors dans `errno`, si `<errno.h>` est inclus).

Remarque :

Le fichier doit bien entendu avoir été ouvert dans un mode qui permet l'écriture, avec `fopen()`. On peut créer des fichiers de données structurées assez simplement (moins facilement qu'en Pascal, mais aucun langage n'est absolument parfait), comme le montre l'exemple ci-dessous.

Exemple :

```
/*-----------------------------------------*/
/* FWRITE.C                                */
/*                                         */
/* Exemple de création de fichier          */
/*                                         */
/* Ce programme créée le fichier TOTO.FIC  */
/* avec trois enregistrements structurés.  */
/*                                         */
/*-----------------------------------------*/
```

```
#include <stdio.h>
#include <stddef.h>
#include <errno.h>

/* petites macros pour simplifier l'écriture : */

#define NOM(X) tampon[(X)].nom
#define PRE(X) tampon[(X)].prenom

/* Le programme */

main()
{
 typedef struct
 {
  char nom[20];
  char prenom[20];
 } PERSONNE;

 PERSONNE tampon[10];
 FILE   *bloc_fic;
 char   *nom_fichier = "TOTO.FIC";
 char   *mode = "w";

 if ((bloc_fic = fopen(nom_fichier,mode)) == NULL)
 {
  puts("Impossible de créer le fichier !");
  exit(1);
 }
 puts("Le fichier TOTO.FIC a été créé avec son bloc de controle.");

 strcpy(NOM(0),"BRANDEIS");
 strcpy(PRE(0),"Pierre");
 strcpy(NOM(1),"PIEROT");
 strcpy(PRE(1),"Francis");
 strcpy(NOM(2),"KAHN");
 strcpy(PRE(2),"Philippe");

 if ( fwrite((void *) tampon,sizeof(PERSONNE),3,bloc_fic) != 3)
 {
  puts("Problème d'écriture dans le fichier !");
  exit(1);
 }

 printf ("%s %s et %s %s remercient %s %s !\n\n",PRE(0),NOM(0),
         PRE(1),NOM(1),PRE(2),NOM(2));
```

```
  if (fclose(bloc_fic))
  {
    puts("Impossible de fermer le fichier !");
    exit(1);
  }
  exit(0);
}
```

Lire dans un fichier avec FCB.

fread()

Syntaxe :

```
#include  <stdio.h>
#include <errno.h>
char     *tampon;
FILE     *fcb;
int      longueur;
int      nombre;
int      resultat;

resultat = fread(tampon, longueur, nombre, fcb);
```

Lit le `nombre` d'éléments de la `longueur` indiquée dans le fichier, qui doit être ouvert auparavant dans un mode de lecture (`r`, `r+`, `a+`). La fonction renvoie le nombre d'éléments lus, ou `-1` si une erreur a eu lieu (le code d'erreur est alors dans `errno`, si `<errno.h>` est inclus).

Remarques :

On peut lire des fichiers de données structurées assez simplement comme le montre l'exemple ci-dessous (suite de l'exemple FWRITE ci-dessus).

Exemple :

```
/*-----------------------------------------*/
/* FREAD.C                                 */
/*                                         */
/* Exemple de lecture de fichier           */
/*                                         */
/* Exécuter l'exemple FWRITE.C avant.      */
/*                                         */
/* Ce programme ouvre le fichier TOTO.FIC  */
/* et y lit trois enregistrements.         */
/*                                         */
/*-----------------------------------------*/

#include  <stdio.h>
#include  <stddef.h>
#include  <errno.h>
```

```
/* petites macros pour simplifier l'écriture : */

#define NOM(X) tampon[(X)].nom
#define PRE(X) tampon[(X)].prenom

/* Le programme */

main()
{
  typedef struct
  {
    char nom[20];
    char prenom[20];
  } PERSONNE;

  PERSONNE tampon[10];
  FILE     *bloc_fic;

  if ((bloc_fic = fopen("TOTO.FIC","r")) == NULL)
  {
    puts("Impossible d'ouvrir le fichier en lecture.");
    exit(1);
  }

  if ( fread((void *)tampon,sizeof(PERSONNE),3,bloc_fic) == -1 )
  {
    puts("Problème de lecture dans le fichier !");
    exit(1);
  }
  printf ("%s %s et %s %s remercient %s %s !\n\n",PRE(0),NOM(0),
          PRE(1),NOM(1),PRE(2),NOM(2));
  exit(0);
}
```

Mettre à jour un fichier d'après le tampon.

fflush()

Syntaxe :

```
#include  <stdio.h>
FILE     *fcb;
int      resultat;

resultat = fflush(fcb);
```

Ecrit le contenu du tampon dans le fichier si celui-ci n'avait pas encore été mis à jour. Cette opération force le vidage du tampon, et assure que le contenu physique du fichier est conforme aux plus récentes opérations d'écriture. Il est nécessaire de procéder à `fflush` de temps à autres pour ne pas risquer la

perte du tampon (sur arrêt du DOS, par exemple). La position dans le fichier n'est pas modifiée. De plus, un bon emploi de la fonction `fflush()` peut accélérer les traitements de façon notoire (les mises à jour du tampon ayant alors lieu de moins en moins souvent, ou plus exactement elles n'arrivent plus de façon imprévue).

La fonction renvoie `-1` si une erreur a eu lieu, et dans ce cas le numéro de l'erreur est dans la variable globale `errno`. SInon, elle renvoie `0` (succès).

Fermer un fichier

fclose()

Syntaxe :

```
#include <stdio.h>
FILE    *fcb;
int     resultat;

resultat = fclose(fcb);
```

Ferme le fichier après avoir écrit les dernières modifications enregistrées dans le tampon, puis libère la mémoire prise par le `fcb`. Suite à `fclose`, le fichier n'est plus ouvert, il faut nécessairement recourir à un nouvel appel de `fopen()` pour le relire ou le modifier (consulter également le manuel Borland sur la fonction `freopen()` qui permet de réouvrir un fichier).

La fonction renvoie `0` si tout s'est bien passé, et `-1` si une erreur est survenue. Dans ce cas le code de l'erreur est placé dans la variable globale `errno` (si `<errno.h>` est inclus).

Positionnement dans un fichier.

fseek()

Syntaxe :

```
#include <stdio.h>
#include <errno.h>
FILE    *fcb;
long    deplacement;
long    resultat;

resultat = fseek(fcb, deplacement, position);
   position :
        SEEK_SET
        SEEK_CUR
        SEEK_END
```

Débute la lecture ou l'écriture dans un fichier à un emplacement donné.

`position` est l'un des trois paramètres ci-dessus, qui représentent respectivement le début du fichier, la position actuelle, et la fin du fichier. La nouvelle position est celle obtenue en appliquant `deplacement` à partir de `position`.

Remarques :

-Le nombre d'octets de `deplacement` par rapport à `position` est spécifié sous la forme d'un entier 32 bits signé.

La fonction renvoie un entier `long` signé également : il prend la valeur `0L` si tout s'est bien passé, `-1L` sinon.

Exemple :

```
/*-----------------------------------------*/
/* FSEEK.C                                 */
/*                                         */
/* Exemple de positionnement dans un fichier */
/*                                         */
/* Exécuter l'exemple FWRITE.C avant.      */
/*                                         */
/* Ce programme ouvre le fichier TOTO.FIC  */
/* et y lit le 3e enregistrement.          */
/*                                         */
/*-----------------------------------------*/

#include <stdio.h>
#include <stddef.h>
#include <errno.h>

/* petites macros pour simplifier l'écriture : */

#define NOM(X) tampon[(X)].nom
#define PRE(X) tampon[(X)].prenom

/* Le programme */

main()
{
   typedef struct
   {
        char    nom[20];
        char    prenom[20];
   } PERSONNE;

   PERSONNE     tampon[10];
   FILE         *bloc_fic;
```

```
    /* avant tout, ouvrir le fichier */
    if ((bloc_fic = fopen("TOTO.FIC","r")) == NULL)
    {
          puts("Impossible d'ouvrir le fichier en lecture.");
          puts("exécutez FWRITE avant, svp...");
          exit();
    }

    /* Se placer sur le 3e enregistrement */
    if (fseek(bloc_fic, -sizeof(PERSONNE),SEEK_END) == -1)
    {
          puts("Problème pour localiser le 3e enregistrement...");
          puts("Exécutez FWRITE avant, svp...");
          exit();
    }

    /* Lire un seul enregistrement */
    if ( fread((void *) tampon, sizeof(PERSONNE),1,bloc_fic) == -1)
    {
          puts("Problème de lecture dans le fichier !");
          puts("Exécutez FWRITE avant, svp...");
          exit();
    }

    printf("J'ai trouvé %s %s dans mon fichier !\n\n",PRE(0),NOM(0));
    exit();
}
```

Conclusion

Turbo C comporte de nombreuses autres fonctions pour travailler sur les fichiers avec FCB. Nous avons uniquement examiné les fonctions essentielles, celles sans lesquelles on ne peut pas travailler. Le reste est une question de raffinement : les quelques fonctions examinées ci-dessus permettent finalement de tout faire.

Le lecteur plus curieux examinera donc avec profit son manuel de référence au sujet des fonctions sur les fichiers, mais nous pensons qu'il est inutile -et probablement néfaste- d'essayer de retenir le fonctionnement d'une trentaine de fonctions lorsque six ou sept suffisent pour 80% des besoins. C'est d'autant plus vrai lorsque l'on essaie d'apprendre le C, et que l'on est quelque peu submergé : il vaut mieux se limiter au strict nécessaire !

Les fonctions que nous venons de voir, à une exception près (`fflush`), ont toutes une sœur jumelle dans la famille des fonctions des fichiers avec handle.

La gestion de fichiers avec les fonctions que nous avons vues est donc identique quelle que soit la méthode utilisée (FCB ou handle). Le choix se portera sur l'utilisation ou non d'un tampon, sachant que les handles sont plus tournés vers l'évolution et l'indépendance du système d'exploitation, et les FCB vers la performance.

CHAPITRE 18

INTERFACE MS-DOS

Le C se caractérise par ses nombreuses possibilités vis à vis du système. Nous avons pu constater la grande influence de Unix sur MS-DOS, et nous allons une nouvelle fois bénéficier de la grande souplesse de C, car il existe un nombre impressionnant de fonctions C permettant d'utiliser directement le système afin de tirer le meilleur parti de MS-DOS.

LA PORTABILITE

Attention, portabilité zéro!

Bien entendu, l'uilisation de la plupart des fonctions que nous allons examiner dans ce chapitre rend les programmes inutilisables sur d'autres machines. Il faut prendre conscience de l'extrême spécialisation de ces fonctions, et savoir les isoler astucieusement, de façon à ne pas avoir trop de travail pour transporter une application par exemple sur un VAX.

Dans la mesure où l'utilisation de ces fonctions est intégrée à une autre fonction, de forme typiquement C, l'ensemble des appels DOS ou BIOS constituera une "couche" de bas niveau, peu encombrante, qu'il suffira alors de modifier pour rendre le programme compatible avec une autre machine. La grande majorité du programme utilisera des appels de fonctions tout à fait classiques, seules les fonctions les plus basses se permettant d'appeler le système.

Vous devez donc particulièrement bien structurer une application devant comporter des appels direct au système, pour que ces appels soient en nombre réduit et trés isolés, faciles à modfier.

Le BIOS : compatibilité douteuse

Enfin, dernière remarque en ce qui concerne la portabilité, nous ne recommandons pas l'utilisation du BIOS. Certains l'ignorent, mais le BIOS n'est absolument pas garanti compatible vers le futur. Il se peut fort bien que certaines fonctions du BIOS disparaissent, que d'autres changent de fonctionnement, suivant les versions de l'IBM PC.

Ce n'est pas le cas des fonctions du DOS, qui, elles, restent en place d'une version de MS-DOS à une autre. On peut donc sans crainte utiliser allègrement toutes les fonctions systèmes du DOS; ces dernières sont d'ailleurs les plus puissantes.

Toutefois, dans certains cas, il peut être nécessaire d'utiliser tout de même les fonctions BIOS, ou pire, directement les ports d'entrées-sorties (contrôleur vidéo, de disque, etc...).

Dans ce cas, vous devez prendre conscience du caractère hautement intransportable de votre application : son fonctionnement ne sera même pas garanti sur tous les compatibles IBM. Il est tout à fait possible que certaines machines réputées très compatibles n'acceptent cependant pas un tel programme. A ce niveau là, il s'agit surtout de chance. Méfiance donc, même le BIOS n'est pas exempt de pièges anti-compatibilité. Tout en sachant que l'application passera peut-être parfaitement sur 99% des compatibles.

Pour garantir un fonctionnement certain sur tout compatible, il est impératif de n'utiliser que les fonctions DOS (c'est à dire celles appelées par le vecteur d'interruption 21H).

QUELQUES FONCTIONS UTILES

Chronomètre

gettime()

Syntaxe :

```
#include <dos.h>
struct time chrono;

gettime(&chrono);
```

La structure `time` est définie comme 4 caractères non signés, dont les noms sont `ti_min`, `ti_hour`, `ti_hund` et `ti_sec`. Ces quatres variables contiennent respectivement la minute, l'heure, les centièmes de seconde, et la seconde de l'heure système.

Exemple :

```
#include <dos.h>
#include <math.h>

#define MAXILOOP 2000
#define TEMPS chrono.ti_min * 6000 + chrono.ti_sec * 100 +
chrono.ti_hund

main()
{
  struct time  chrono;
  long     i ;
  long     debut,fin;
  double    x=0.1;
```

```
  printf("Je vais chronométrer %d calculs de sinus.",MAXILOOP);
  gettime(&chrono);
  debut = TEMPS;
  for (i=0 ; i<MAXILOOP; i++) x=sin(x);
  gettime(&chrono);
  fin = TEMPS;
  printf("Il m'a fallu %3.2lf secondes.\n",(fin-debut)/100.0);
}
```

Répertoires de MS DOS

getcurdir()

Syntaxe :

```
#include <dir.h>
char     *repertoire;
int      disque;
int      resultat;
repertoire = (char *) malloc(MAXDIR);

resultat = getcurdir(disque,repertoire);
```

Renvoie dans la chaîne `repertoire` le chemin du répertoire actuel. `disque` contient 0 pour le disque par défaut (actuel), 1 pour le lecteur `A:`, 2 pour `B:`, etc.

Le nom renvoyé ne comporte ni le nom du disque, ni un "\" de début. Si la fonction s'est correctement effectuée, elle renvoie `0`, sinon `-1`.

Exemple :

```
#include <dir.h>
#include <stddef.h>
#include <alloc.h>

main()
{
   char *rep;
   int  drive;

   drive = 3;  /* disque dur C: */
   if ( (rep = (char *) malloc(MAXDIR+3)) == NULL)
   {
        puts("Impossible d'allouer de la mémoire");
        exit();
   }
```

```
   /* Ci dessous, "C:\\" représente  C:\ car le backslash */
   /* est le préfixe des caractères non imprimables */
   strcpy(rep,"C:\\");
   /* Récupérons le nom du répertoire sans écraser celui*/
   /* du disque */
   if (getcurdir(drive,rep+3))
   {
        puts("Problème...");
        exit();
   }
   printf("Le répertoire actuel est %s\n",rep);
}
```

chdir()

Syntaxe :

```
#include <dir.h>
char     *repertoire;
int      resultat;

resultat = chdir(repertoire);
```

Change de répertoire courant. La chaîne contenant le nom du nouveau répertoire peut inclure le nom du disque. Le slash "/" peut remplacer le backslash, celui-ci devant être doublé dans la chaîne "\\".

Si le répertoire est trouvé, la fonction renvoie `0` sinon elle renvoie `-1`.

Exemple :

```
#include <dir.h>
#include <stddef.h>
#include <alloc.h>

main()
{
  char  *rep = "C:/BASICA";

  if (chdir(rep))
  {
    puts("Problème...");
    exit();
  }
  printf("Le répertoire actuel est maintenant %s\n",rep);
}
```

Exécution de programmes externes

execl(),execlp(),exec...()

Syntaxe :

```
#include <process.h>
#include <stdio.h>
char     *nom_programnme;
char     *arg0;
char     *arg1;
. .
int      resultat;

resultat = execl(nom_programme, arg0, arg1, ... NULL);
```

Exécute le programme dont le nom est passé, avec les arguments précisés. Le nom du programme doit comporter le chemin d'accès. Pour utiliser le chemin de recherche PATH de MS-DOS, servez-vous de la fonction `execlp`,qui a la même syntaxe (inutile dans ce cas d'indiquer le nom du programme). Les arguments sont optionnels (sauf `NULL` qui indique la fin de leur liste) : ce sont les paramètres passés au programme.

La fonction ne renvoie un résultat que si une erreur a eu lieu, car `execl` ou `execlp` ne rendent pas la main. Le programme appelant est écrasé et l'exécution du processus suivant est demandée. Ceci suppose qu'il reste suffisamment de mémoire disponible. On peut préciser n'importe quel programme, qu'il soit de type EXE ou COM. On peut aussi préciser une commande DOS ou un fichier de commandes BAT, mais dans ce cas il faut passer par COMMAND.COM et indiquer le nom de la commande ou du fichier de commandes en paramètre 1 (cf exemple EXECLP.C ci-dessous). Notez que l'interpréteur de commandes doit être appelé avec le paramètre /C pour éviter qu'il ne reste résident après exécution de la commande ou du fichier BAT. Il hérite bien entendu de l'environnement, notamment du chemin de recherche PATH.

Note importante : sous MS-DOS, `arg0` est toujours le nom du programme lui-même. Voir exemple ci-dessous.

Exemple :

```
#include <stdio.h>
#include <process.h>

main()
{
  char  *nom = "COMMAND.COM";

  puts("Attention, je vais exécuter un sous-processus ...");
```

```
  if (execlp(nom,nom,"/C","DIR","/W",NULL))
  {
    puts("J'ai rencontré un problème...");
    exit(1);
  }
}
```

APPEL DES FONCTIONS DE MS-DOS

intdos()

Syntaxe :

```
#include <dos.h>
union REGS       entrees;
union    REGS    sorties;
int      resultat;

resultat = intdos(&entrees,&sorties);
```

Exécute une fonction interne de MS-DOS (interruption de vecteur 21H). Les données passées en paramètres sont dans la variable `union entrees`. Cette `union` est constituée des variables suivantes :

```
int x.ax, x.bx, x.cx, x.dx, x.di, x.si, x.cflag, x.flags;
```

ou les registres `x.?x` peuvent aussi être représentés par :

```
char h.al, h.ah, h.bl, h.bh, h.cl, h.ch, h.dl, h.dh;
```

Il faut comme pour une fonction DOS classique remplir `entrees.h.al` avec le code de la fonction demandée, et les registres voulus de `union entrees` avec les valeurs souhaitées.

Une fois la fonction exécutée, le résultat est placé dans l'`union sorties`. Celle-ci peut être consultée de la même façon. Le drapeau Carry est spécialement copié dans `sorties.x.cflag`. S'il est positionné, c'est qu'une erreur a eu lieu.

Notez que `entrees` et `sorties` peuvent être la même variable, si l'on ne désire pas conserver les valeurs entrées.

Veuillez vous reporter à un manuel technique MS-DOS pour connaître les codes et paramètres des fonctions internes de MS-DOS. Nous conseillons également "Clefs pour IBM PC/AT et compatibles" (chez le même éditeur)I qui résume la totalité de ces fonctions (bien d'autres livres se penchent sur la question, par ailleurs : consultez la bibliographie à ce sujet pour en savoir plus).

Exemple :

```
#include <dos.h>

main()
{
  union REGS regs;
  int   Rax;
  char  mois[12][11]= {"Janvier","Février","Mars","Avril",
              "Mai","Juin","Juillet","Aout",
              "Septembre","Octobre","Novembre","Décembre"};
  char  jour[7][10]  = {"Dimanche","Lundi","Mardi","Mercredi",
              "Jeudi","Vendredi","Samedi"};

  regs.h.ah = 0x2A; /*fonction 2a = lecture date système */
  intdos(&regs,&regs);
  if (regs.x.cflag)
  {
   puts("Erreur dans l'exécution de la fonction DATE !!!");
   exit(1);
  }
  printf("Aujourd'hui : %s %2d %s %4d.\n",jour[regs.h.al],
                        regs.h.dl,
                        mois[regs.h.dh],
                        regs.x.cx);
}
```

CONCLUSION

Il existe d'innombrables autres fonctions extrêmement intéressantes dans Turbo C. Mais comme nous l'avons déja souligné, le but de ce livre n'est pas de vous emmener au sommet de la pyramide des grands maîtres du C ! Aussi, commencez déjà par bien assimiler les exemples de ce chapitre, en les intégrant par exemple dans vos programmes, ou en les modifiant. Nous avons seulement effectué un très large tour de l'horizon dans ce chapitre. Il vous reste à découvrir `int86x`, la gestion de processus parentaux, l'interfaçage direct de routines en assembleur, les appels BIOS, l'examen et modification direct de la mémoire par ... `peek()` et `poke()` (tiens tiens...), etc...

Nous aurions bien entendu pu remplir ce livre d'astuces et d'outils utilisant le systême : Turbo C facilite tellement leur mise au point comparé à l'assembleur, que c'en est véritablement déconcertant lorsqu'on pratique ce dernier par ailleurs. Et notre but était surtout, dans ce chapitre, de démontrer l'extraordinaire polyvalence de Turbo C, qui permet aussi bien de gérer un fichier ou un texte que de reprogrammer l'interpréteur de commandes DOS ou la gestion des saisies de l'utilisateur.

N'oubliez pas que l'utilisation des fonctions DOS et système recèle une grande puissance, mais qu'elle interdit toute compatibilité instantanée avec les autres systèmes d'exploitation.

Ces outils Turbo C peuvent cependant être plus qu'utiles pour la mise au point, par exemple, d'outils systèmes spécifiquement MS-DOS. Ces fonctions vous permettent alors de travailler au niveau du système sans recourir à l'assembleur. Ce chapitre vous a permis de constater que la mise en œuvre des fonctions les plus internes du système est d'une facilité déconcertante avec Turbo C.

Ce qui enrichit notre refrain concernant la puissance et la souplesse du langage C par de nouveaux vers. Il semble décidément que Turbo C sache tout faire !

CHAPITRE 19

LES PIEGES

Ce chapitre est présent dans tous les ouvrages d'initiation au langage C. Ce dernier autorise tellement de choses, que les erreurs de début sont inévitables. Le but ici est donc de faciliter vos premiers pas en mettant en évidence les pièges dans lesquels, c'est inévitable, vous tomberez forcément au début. Nous les connaissons bien, puisque nous sommes nous-mêmes tombés dans ces pièges auparavant.

Pour aller un peu plus loin, nous nous sommes également permis de vous expliquer comment mettre en œuvre toute la puissance d'aide de Turbo C, grâce aux warning. Vous trouverez donc dans la dernière partie de ce chapitre une explication pour mettre déclencher les warnings, et la liste de tous ces warnings, avec leur signification probable.

Nous espérons que vous n'aurez pas trop besoin de ce chapitre une fois que vous l'aurez lu. Mais si vous y passez trop de temps à votre goût, ne vous inquiétez pas : C est le type même du langage difficile au premier abord car touffu, mais qui s'apprend trés vite. Les erreurs sont notamment rapidement comprises car elles sont caractéristiques, et en très peu de temps vous les reconnaîtrez tout de suite.

ARITHMETIQUE ET EXPRESSIONS

Comme nous l'avons expliqué depuis le début, C autorise une telle souplesse dans les calculs des expressions, qu'elles sont par essence même une source inévitable d'erreurs et de mauvais fonctionnements.

Les parenthèses (erreur de compilation).

D'avance, nous pouvons prédire que vous oublierez forcément des parenthèses un peu partout. C'est tout à fait normal, car c'est le genre d'erreur qui frappe même des programmeurs expérimentés du C.

Notre conseil est donc le suivant : ne programmez pas sur une même ligne une expression trop complexe.

N'oubliez pas que le compilateur ne distingue pas les fins de lignes. Profitez-en donc pour couper vos expressions, les aligner proprement, les entourer de parenthèses qui se font écho logiquement. Le compilateur s'y retrouvera toujours, lui. Alors facilitez-vous plutôt la vie. Et puis finalement, le programme généré sera le même. Regardez la différence de clarté des deux expressions ci-dessous :

```
calcul=((valeur1*100)+(test2/100.0*ttc)-exp(resultat)/PI));

calcul = (
           (valeur1  * 100)
          +(test2 / 100.0 * ttc)
          - exp(resultat) / PI
         );
```

Et de plus, la première expression comporte une parenthèse de trop. Ne prétendez pas que cela saute aux yeux, tout de même !

Certes, la première est plus jolie, c'est un fait. Mais n'oubliez pas que c'est vous qui mettez le programme au point : et de toute évidence, la seconde présentation laisse beaucoup moins de place à l'erreur d'étourderie. De toutes façons, nous insistons la dessus, le code généré sera exactement le même, et les performances seront identiques.

Les conversions automatiques

Nous allons sans doute nous répéter à ce sujet. Mais n'oubliez pas les points suivants, si votre programme fonctionne bizarrement ou si vous devez rédiger une expression calculatoire complexe :

- La conversion automatique ne doit se faire que dans le sens d'un nombre d'octets croissants : `char` vers `short`, `short` vers `long`, `long` vers `float`, `float` vers `double`. Jamais dans l'autre sens, à moins alors de tester auparavant les valeurs pour être sûr de ne pas provoquer un dépassement de capacité, car en C, cela n'est pas une erreur d'exécution : la variable incapable de recevoir la valeur effectue alors un choix dans les octets, choix qui, soyons-en sûrs, ne correspond que rarement à celui du développeur.

- N'oubliez pas que les calculs en nombres réels sont effectués en type `double`, quoi qu'il arrive. Attention donc à la récupération d'un résultat dans une variable `float`, par exemple. Usez et abusez du casting.

- Attention aux innocentes divisions entières invisible: `1/3` vaut intrinsèquement `0`, car la division travaille sur deux nombres entiers, et donc donne un résultat entier. Ajoutez "`.0`" aux constantes si vous voulez un résultat réel : `1.0` au lieu de `1`, `-34.0` au lieu de `-34`, et ainsi de suite.

- Attention à l'ordre de priorité des opérateurs. Le mieux est d'entourer toute expression simple (opérande opérateur opérande) avec des parenthèses. L'ordre d'évaluation par défaut se trouve page 145 du manuel utilisateur Borland (et en annexe dans ce livre).

Expressions composées

Attention aussi aux expressions composées. Si vous modifiez une variable dans une des expressions, elle possèdera sa nouvelle valeur dans les suivantes. Exemple :

```
resultat = ((i=1),i++,i--);
```

Ceci donne tout d'abord la valeur 1 à i. Puis, i est incrémenté et vaut 2, et enfin i est décrémenté, donc vaut de nouveau 1.

Le programmeur qui ne possède pas encore bien le principe des expressions multiples aura tendance à croire que le programme assigne 1 à i, puis calcule i+1, puis i-1, ce qui donne 0. C'est bien entendu faux .

Deux conseils s'imposent au sujet des opérateurs ++ et -- :

- Etant donné que leur effet est immédiat et que l'ordre d'évaluation des expressions peut changer d'un compilateur à un autre, ne les utilisez surtout pas à l'intérieur d'une expression composée si l'une des autres expressions utilise sa valeur. L'effet est imprévisible car vous ne savez pas si la variable sera utilisée avant que (variable++) soit évalué, ou après, par exemple. Ce qui interdit toute prévision pour le calcul.

- Ne les utilisez surtout pas, jamais, sous aucun prétexte, en tant que paramètre d'une fonction. Surtout avec les fonctions de la bibliothèque. En effet, la plupart de ces foncions sont en fait des macros. Nous n'avons pas décrit les macros pour éviter ces pièges. Ce sont des #define qui peuvent recevoir un paramètre ou plusieurs. Voici un exemple :

```
#define ABS(X) ( ((X)>0) ? (X) : (-(X)) )
```

Dans ce cas, par exemple, l'expression ABS(i++) sera remplacée lors de la compilation par :

```
( ((i++)>0) ? (i++) : (-(i++)) )
```

Inutile de dire que dans tous les cas, i++ est exécuté deux fois, ce qui n'est visiblement pas l'effet souhaité par l'appel ABS(i++).

Opérateurs d'affectation et logiques

Une erreur fréquente est de confondre == avec =, ou une affectation avec un test. C'est d'autant plus fréquent, que les deux peuvent avoir un sens.

Par exemple la séquence suivante place la valeur 2 dans i :

```
i=1;
...
if (i==1) i=2;
```

Et le plus grave, c'est que celle-ci aussi :

```
i=1;
...
if (i=1) i=2;
```

N'oubliez jamais que toute expression donnant un résultat différent de zéro a une valeur logique VRAIE. Si vous confondez vraiment souvent == avec =, vous pouvez utiliser le #define afin de remédier astucieusement au problème :

```
#define EGALE ==
#define DEVIENT =
#define SI if
```

Par la suite, notre premier exemple devient :

```
  i DEVIENT 1;
   ...
  SI (i EGALE 1) i DEVIENT 2;
```

Vous ne pouvez plus vous tromper !

Maintenant, est-ce plus lisible, c'est à vous de juger. Vos expressions risquent de devenir éléphantesques !

Dans ce cas, autant simplifier un peu :

```
#define EGAL ==
#define DIFF !=
i = 1;
if (i EGAL 1) i=2;
```

Dans tous les cas, il vous appartiendra de redéfinir ou non certains mots-clé qui ne vous plaisent pas. Mais n'oubliez pas, n'en faites pas trop, car vous risquez de ne plus vous y retrouver du tout en C standard. Limitez ce genre de manipulations aux cas vraiment épineux que vous n'arrivez pas à maîtriser.

POINTEURS CHAINES ET RESERVATION MEMOIRE

L'utilisation des pointeurs est aussi sujette à de nombreuses erreurs. La grosse différence avec les erreurs d'expressions, c'est que la mauvaise utilisation de pointeurs peut provoquer une catastrophe ou , au mieux, un arrêt complet du systême.

Bien entendu, souvent, et c'est là le piège, rien d'évident ne surviendra avant un certain temps.

Supposons par exemple que vous procédiez dans un programme à ce qui suit :

```
double *x;
...
*x = 3.14159269;
```

L'erreur est commise. Et comme l'explique Borland dans son manuel, le plus grave peut-être, c'est que cela peut très bien ne rien provoquer de grave.

L'erreur, c'est d'avoir modifié `*x`, c'est à dire la zone mémoire pointée par le pointeur `x`, alors que justement ce pointeur n'a pas reçu d'adresse valide, et que par conséquent, il pointe sur un endroit tout à fait quelconque de la mémoire. Cela peut aussi bien être en plein milieu de la mémoire libre (rien de grave ne survient), qu'en plein écran (d'étranges symboles apparaissent sur celui-ci) ou, plus spectaculaire sans doute, dans une zone réservée au système (blocage, ou grosse anomalie).

Il est donc impératif, lorsque l'on utilise des pointeurs et le contenu des zones qu'ils adressent, de s'assurer qu'une valeur leur a été assignée, par exemple avec `malloc()`, et qu'une zone mémoire libre leur a été réservée. L'exemple ci-dessus doit devenir :

```
double  *x;
...
if ( x = (double *) malloc(sizeof(double)) ) == NULL)
{
   ... pas d'espace alloué, traiter cette erreur ...
}
*x = 3.14159269;
```

Notez le test incluant l'appel de `malloc()`, qui permet de s'assurer (deux précautions valent mieux qu'une) que le pointeur `x` a effectivement reçu une adresse, et que par conséquent une zone mémoire lui a été allouée.

Bien entendu, cette réservation mémoire n'est effectuée qu'une seule fois, sans quoi on risque de saturer celle-ci, et surtout on change de valeur pointée à chaque fois !

Si vous voulez libérer une zone mémoire, utilisez la fonction `free()`. Consultez le manuel Borland à ce sujet. Cette libération est en réalité plus utile pour les listes chaînées, B-arbres et autres structures de données dynamiques, sujet que nous n'avons pas abordé dans cet ouvrage (il faudrait un livre entier pour en parler, et c'est de la programmation déjà complexe).

LISIBILITE

Au risque (c'est même une certitude) de nous répéter, la lisibilité d'un programme est une chose vitale en C.

Non pas que le compilateur aime les belles imbrications de boucles : en réalité, il ne les voit pas.

Mais vous avez pu constater que C ne brille pas par sa clarté intrinsèque : la plupart des opérateurs ressemblent à des hiéroglyphes, et les fonctions de la librairie souffrent toutes des limitations anciennes concernant la longueur des identificateurs.

Par exemple, `strcpy` est sans doute moins évocateur que `string_copy`. Mais les fonctions datent de l'époque où les identificateurs n'avaient que 6 caractères significatifs, et leur clarté s'en ressent.

Puisque le langage n'est ni très clair ni explicite, c'est donc au programmeur de s'arranger pour que les listings le soient tout de même.

Le premier réflexe, assez passionel il est vrai, est d'utiliser l'efficacité du C pour rédiger des lignes de programme extraordinaires, pour obtenir un programme potentiellement génial et illisible. Incontestablement, cela a de l'allure.

Mais si par malheur il y a une cause de mauvais fonctionnement ...

Soyons brefs : un listing qui vous paraît génial aujourd'hui vous paraitra déja illisible demain. C'est le propre du langage C.

Commentaires, imbrications cohérentes, découpage des expressions ou instructions compliquées sur plusieurs lignes, n'hésitez pas à payer de votre personne.Le listing sera plus long, mais vous y gagnerez au bout du compte. N'oubliez pas, encore un leitmotiv, que c'est vous qui mettez le programme au point et non le compilateur. Il est donc inutile de lui faciliter la vie (ou de croire le faire, car en fait cela ne change absolument rien pour lui) en supprimant les blancs, les indentations, les tabulations, les commentaires. Tout ce que vous pouvez à la rigueur gagner, c'est un peu de place sur le disque.

Une remarque en passant concernant les commentaires : vous avez pu remarquer qu'un listing C est déjà un bel enchevêtrement de parenthèses, d'étoiles *, de ++, de soulignés _ , de &, de \n " : ? etc...

Est-il vraiment plus clair d'y ajouter des /* et des */ ?

En fait, non. Si vous mettez des commentaires n'importe où et sous n'importe quelle forme, certes le programme sera documenté, ce qui est un bon point, mais il n'en sera pas plus lisible, bien au contraire.

Il est donc essentiel d'adopter une démarche cohérente à ce sujet.

- Ne pas hésiter à placer des lignes vides entre les blocs importants du programme.

- Pour les commentaires importants, expliquant le rôle d'une portion de programme par exemple, utiliser un cadre :

```
/*-----------------------------------*/
/* CECI EST UN COMMENTAIRE IMPORTANT */
/*-----------------------------------*/
```

- Pour les commentaires locaux, expliquant ce que fait telle ou telle ligne, par exemple, réserver le tiers le plus à droite de l'écran. Aligner systématiquement les débuts de commentaires (par exemple sur la colonne 60), et si possible les fins. Comme en assembleur, en quelque sorte :

```
/*--------------------------------------*/
/* Une belle boucle .....               */
/*--------------------------------------*/

 for (i=0 ; i<10 ; i++)              /* 10 boucles sur i       */
      printf("%d\n",i);             /* Affiche i, à la ligne. */
```

- Eventuellement, mais cela dépend de vos goûts, redéfinissez quelques symboles importants du C pour clarifier les listings. Le plus approprié (surtout sur un clavier français) est de procéder aux définitions suivantes :

```
#define begin {
#define end }
```

Mais vous pouvez vous en passer. Dans tous les cas, n'oubliez pas que cela peut tout aussi bien vous empêcher de progresser correctement en C (surtout si vous pensez trop Pascal : nous avons fait l'erreur au début, nous aussi !).

LES WARNINGS DE TURBO C

Une erreur vous résiste : impossible de savoir d'où elle provient...

Merci Borland, car le compilateur Turbo C intègre de nombreuses possibilités d'aide dans ce sens.

Outre les innombrables messages erreurs qui peuvent survenir à la compilation, Turbo C inclut un module `LINT`, c'est à dire un programme vérificateur, qui se charge de vérifier tous les points épineux de votre programme, et indique quels sont ceux qui pourraient provoquer une erreur.

Il est important de comprendre que ces warnings ne constituent aucunement une erreur : le programme est parfaitement exécutable. Mais il est souvent intéressant de consulter les messages envoyés par le module `LINT`, car parfois ils permettent de détecter une erreur ... avant qu'elle ne soit mise en évidence par un plantage général.

Mise en oeuvre des warnings.

Turbo C est livré avec un certain nombre de ces warnings actifs. Mais d'autres sont inactifs, il faut les positionner si vous désirez profiter de leurs services.

Notez que l'utilisation des warnings ralentit légèrement la compilation. D'autre part, ils ne sont réellement efficaces, pour la plupart, que si vous utilisez soit les prototypes (mais nous avons dit de les utiliser systématiquement...), soit des déclarations de fonctions de type ANSI (et non pas K&R).

Pour arriver au menu des activations de warnings, vous devez frapper successivement les touches suivantes :

`Alt-O` (pour arriver au menu Options)

`C` (sous-menu Compiler)

`E` (sous-sous-menu Errors)

`D` (pour indiquer "Display warnings ON" si indique "OFF")

`P A C ou L` : les 4 menus d'activation.

Suivant le dernier menu choisi, Turbo C propose un certain nombre de warnings (6 ou 7) regroupés par sujet. Le menu `P` traite des éventuels problèmes de Portabilité, le menu `A` des violations du standard ANSI, le menu `C` des erreurs les plus courantes, et `L` des erreurs tout de même rares. Une fois

dans le menu, l'appui sur la touche associée au warning modifie l'état de son activation. Lorsque les activations vous plaisent, il suffit de remonter par <escape>. N'oubliez pas de sauver les nouvelles options par Alt-O suivi de S, ou vous devrez repositionner les warnings au prochain appel de Turbo C.

Nous allons examiner les warnings un par un.

Le menu des warnings de Portabilité (menu P)

`A: Non-Portable pointer conversion`

Conversion de pointeur non portable.

Indique que l'expression dans une instruction `return()` dans une fonction n'est pas du même type que celui déclaré dans l'en-tête ou le prototype de cette fonction. Généralement cela n'arrive qu'avec les résultats de type pointeurs, puisque nous avons vu que les autres types étaient très souvent compatibles. Toutefois une fonction qui renvoie une constante `0` (par exemple `NULL`) au lieu d'un pointeur fonctionne correctement, car les constantes nulles sont dans ce cas formatées au type pointeur par le compilateur.

`B: Non-Portable pointer assignement.`

Assignation de pointeur non portable.

Indique qu'une variable pointeur reçoit une valeur d'un type différent, ou l'inverse. Ce problème n'en est pas un avec les constantes nulles. Si toutefois c'est ce que vous désiriez vraiment faire, il vaut sans doute mieux placer un casting pour vous débarrasser du warning et avoir l'esprit serein.

`C: Non-Portable pointer comparison.`

Comparaison de pointeur non portable.

Indique que dans un test, un pointeur a été comparé à une valeur de type différent (et non nulle). Encore une fois, il est plus prudent d'utiliser un casting pour être certain de ce que fait le test.

`D: Constant out of range in comparison.`

Constante hors limites dans une comparaison.

Indique que dans une comparaison, une constante a une valeur qui ne signifie rien comparée au type de l'autre membre de la comparaison. Par exemple, comparer une valeur de type `char` à la valeur `10000` n'a pas de sens. Peut-être faut-il utiliser un suffixe pour la constante, ou bien il peut s'agir d'une faute de frappe.

`E: Constant is long.`

Constante de type long.

Indique que le compilateur a rencontré une constante entière dont la valeur dépasse les limites du type `short/int`, et que la constante a été traitée

comme si elle était de type `long`. Il vaut mieux vérifier que la constante possède la valeur souhaitée (faute de frappe ?) et ajouter le suffixe `U` (pour non signée, par exemple si l'on souhaite comparer un entier `short` à `65000`, mettre `65000U`) ou `L` suivant le cas.

```
F: Conversion may lose significant digits.
```

La conversion peut provoquer une perte de chiffres significatifs.

Dans une opération visant à passer d'un type `long` à un type `int` (tous deux signés ou non), il est possible, suivant la machine et le compilateur utilisé, que la conversion automatique provoque une réduction du nombre d'octets. C'est le cas sur l'IBM PC, puisque le type `int` occupe 16 bits, et `long` 32. Ce problème peut nuire à la portabilité, c'est pourquoi il est signalé.

```
G: Mixing pointer to signed and unsigned char.
```

Conversion de pointeur de caractères signés en non signés, ou l'inverse.

Indique que le programme a implicitement converti un pointeur de type `char` en pointeur de type `unsigned char`, sans passer par un cast. Sur l'IBM PC avec le 8086, c'est généralement sans danger, mais il vaut mieux remédier au problème et utiliser un casting.

Le menu des violations ANSI (Menu A)

```
A: 'ident' not part of structure
```

'`ident`' n'appartient pas à une structure.

Indique que dans une expression, le champ indiqué '`ident`' n'appartient pas à la structure indiquée à gauche du point "." ou de la flèche "->", ou bien que l'identificateur situé à gauche du point ou de la flèche n'a rien à voir avec une structure.

```
B: Zero length structure.
```

Structure nulle.

Indique que la structure déclarée a une taille nulle. Toute utilisation se solde donc par une erreur, puisqu'elle ne contient aucun octet.

```
C: Void functions may not return a value
```

Une fonction `void` ne peut pas renvoyer de résultat.

Une fonction de type `void` contient une instruction `return()`. Le principe même du type `void` est justement de ne rien renvoyer. La valeur entre les parenthèses du `return()` est alors ignorée.

`D: Both return and return of a value used.`

Utilisations différentes de return dans une fonction.

Indique que certaines des instructions `return()` de la fonction renvoient un résultat, d'autre non. C'est généralement une erreur ou un oubli.

`E: Suspicious pointer conversion.`

Conversion de pointeur suspecte.

Le compilateur a rencontré une conversion d'un pointeur qui lui donne un autre type. Il vaut mieux dans ce cas placer un casting (si ce n'est pas une faute de frappe ou une erreur bien sûr !).

`F: Undefined structure 'ident'.`

Structure '`ident`' non définie.

La structure indiquée a été utilisée dans le programme mais n'est pas déclarée dans le source. Il manque peut-être une inclusion de fichier, ou bien la structure est déclarée/utilisée avec une faute de frappe, ou bien elle a tout simplement été oubliée.

`G: Redefinition of 'ident' is not identical.`

La nouvelle définition de '`ident`' n'est pas identique à l'ancienne.

Une définition de macro existante n'est pas identique à la précédente : le nouveau texte remplace l'ancien qui était en cours. Vérifier que le texte de la définition est bien correct.

Le menu des erreurs courantes (menu C).

`A: Function should return a value.`

La fonction devrait renvoyer une valeur.

La fonction n'est pas de type `void` ni `int`, mais ne renvoie pas de valeur par `return()`. C'est sans doute une erreur, car si la fonction ne doit rien renvoyer il vaut mieux utiliser le type `void`. Le type `int` permet de ne rien renvoyer no plus car le type `void` est nouveau, et auparavant il était simulé par `int`.

`B: Unreachable code.`

Portion de code inaccessible.

Une portion de code suit une instruction `break`, `goto`, `return` ou `continue` dans le même bloc. Ce code ne sera jamais exécuté.

`C: Code has no effect.`

Code sans effet.

Une portion de code ne correspond à rien. Par exemple, la ligne suivante :

```
a == 3;
```

Elle est syntaxiquement correcte mais ne fait rien. Il y a sans doute une faute de frappe ou un oubli.

```
D: Possible use of 'ident' before definition.
```

Utilisation possible de 'ident' avant son initialisation.

Warning assez fréquent : indique que la variable 'ident' est peut-être utilisée avant d'avoir été initialisée. Il se peut aussi que cela n'ait aucune importance. Cela en a une si la variable est de type pointeur, puisque ce warning peut alors signifier que le programme n'a pas réservé de zone mémoire, par exemple pour un pointeur de variable dynamique ou de chaîne.

```
E: 'ident' is assigned a value wich is never used.
```

'ident' reçoit une valeur qui n'est jamais utilisée.

Indique que la variable reçoit une valeur dans une assignation, mais n'est utilisée nulle part ailleurs. Ce n'est pas forcément une erreur. Vérifier simplement que la variable est vraiment utile, ou qu'elle n'a pas été confondue avec une autre.

```
F: Parameter 'ident' is never used.
```

Le paramètre 'ident' n'est pas utilisé.

Le paramètre indiqué n'est pas présent dans le corps de la fonction. Il se peut qu'il y ait une faute de frappe dans son nom, ou qu'une variable locale déclarée dans la fonction ait le même nom (auquel cas elle rend le paramètre invisible).

```
G: Possibly incorrect assignment.
```

Assignation peut-être incorrecte.

Ce warning intervient quand le compilateur rencontre un opérateur d'assignation comme opérateur principal dans une expression conditionnelle comme `if`, `while` ou `do-while`. Il s'agit souvent d'une faute de frappe (= au lieu de ==, par exemple). Sinon, mettre des parenthèses et comparer à zéro pour obtenir le même effet en supprimant le warning.

Le menu des erreurs rares (menu L).

```
A: Superfluous & with function or array
```

`&` inutile

Un opérateur `&` a été placé devant un nom de tableau ou de fonction passé en paramètre, ce qui est inutile puisque ces identificateurs sont déjà des adresses eux-mêmes. L'opérateur est simplement ignoré par le compilateur.

`B: 'ident' declared but never used.`

'`ident`' est déclaré mais jamais utilisé.

La variable en question n'est pas utilisée à l'intérieur de son bloc de visibilité. Ou bien elle est inutile, ou bien le programmeur a oublié de programmer la portion qui la concernait !

`C:Ambiguous operators need parentheses.`

Opérateurs ambigus, utiliser des parenthèses.

Ce warning survient lorsque deux opérateurs de décalage (<< ou >>), relationnels (!=, ==, etc...) ou logiques (~, !, |, etc.) ont été utilisés ensemble sans parenthèses. Il est préférable de mettre celles-ci pour explicitement préciser la hiérachie des opérations.

`D: Structure passed by value.`

Structure passée par valeur.

Une structure a été passée en paramètre à une fonction sans utiliser l'opérateur `&`. Avec d'anciens compilateurs, une structure ne pouvait pas être passée par valeur, il fallait obligatoirement passer l'adresse de cette structure. Avec Turbo C (norme ANSI oblige), le passage des structures par valeur est autorisé, mais il s'agit probablement quand même d'une erreur, car le plus souvent on aura uniquement oublié de taper le `&` pour le paramètre.

`E: No declaration for function 'ident'`

La fonction '`ident`' n'est pas déclarée.

Une fonction est utilisée avant d'être déclarée. Si la fonciton est de type `int`, ce n'est pas grave puisque c'est le type par défaut. Par contre si c'est un autre type, il faut soit placer la fonction avant, soit utiliser un prototype (ce que nous recommandons chaudement).

`F: Call to function with no prototype.`

Appel d'une fonction sans prototype.

Ce warning indique que la fonction est appelée alors qu'elle n'a pas de prototype. Cela peut arriver pour `printf` et `scanf` si `#include <stdio.h>` n'est pas présent, mais la compilation et le link se déroulent normalement. Rappelons quand même que l'utilisation de prototypes est très profitable.

ANNEXE A

PROGRAMMATION STRUCTUREE : UN EXEMPLE

Nous allons donner l'exemple du calcul d'un taux de rentabilté interne. Ce petit outil financier illustrera le chapitre 10.

QU'EST-CE QUE LE TAUX DE RENTABILITE INTERNE?

Enonçons le problème. Vous disposez, sur un compte bancaire avec intérêt, d'une certaine somme en début d'année. Vous allez, au long de cette année, déposer ou retirer certaines sommes d'argent à chaque fin de mois.

La question est la suivante : sachant que l'argent déposé rapporte un taux `i`, et connaissant les dépenses et les gains de l'année, l'équilibre est-il assuré ? Ou plutôt, à partir de quel taux d'intérêt l'équilibre sera-t-il assuré, pour ces mêmes données ?

Le taux ainsi calculé se nomme "taux de rentabilité interne". Il s'agit d'un taux minimal: si le taux de rendement votre compte est inférieur, vous finirez l'année avec un compte débiteur. Plus il sera supérieur au taux de rentabilité, plus il vous restera d'argent en fin d'année. C'est mathématique.

Le principal problème est que les gains croissent avec un même taux, mais les plus récents ne rapportent pas autant que les plus anciens. De plus, le retrait d'argent vient perturber les calculs.

La seule méthode pratique consiste en une suite d'itérations, afin de déterminer le taux pour lequel l'équilibre final est proche de zéro (avec une précision que nous choisirons).

Le dernier problème est le suivant : comment calculer justement l'équilibre final ? Pour cela, il existe une formule toute prête :

Si F0 est la quantité monétaire de départ (le contenu du compte initial), et F1, F2, jusqu'à F12, les quantités ajoutées ou retranchées à ce compte à chaque fin de mois (on appelle ces valeurs des FLUX), et "i" le taux d'intérêt, l'équilibre final du compte sera le résultat de la formule suivante :

$$F0+\sum_{j=1}^{12}\left(\frac{Fj}{(1+i)^{j}}\right)$$

Cette formule illisible est un standard des calculs financiers : on la rencontre un peu partout sous diverses formes.

ANALYSONS LE PROBLEME

Maintenant que nous connaissons le problème et la formule, nous devons analyser la façon dont allons programmer le tout.

Notre première étape est l'analyse suivante :

Etape 0 :Le programme principal.

1) Stocker par un moyen quelconque (par exemple saisie dans un tableau) les différents flux, F0 jusqu'à F12.

2) Calculer le taux de rentabilité interne inconnu, "i".

3) Procurer le résultat d'une façon quelconque (affichage, impression...)

4) Fin du travail.

Les étapes 1 et 3 ne sont guères sujettes à analyse en profondeur : nous savons comment stocker des valeurs dans un tableau (nous appellerons celui-ci `flux`, il comportera les cases 0 à 12).

Par contre l'étape 2 n'est pas instantanément programmable : nous devons l'analyser un peu plus en profondeur.

Pour calculer le taux, nous allons procéder à une dichotomie, c'est-à-dire une recherche dans un intervalle par affinement progressif de cet intervalle. Le taux est en effet compris entre 0 et 1 (un taux d'intérêt négatif ne présente pas vraiment d'intérêt, puisque plus vous déposeriez d'argent, moins il y en aurait sur le compte !). La dichotomie est une méthode classique qui a fait ses preuves et qui fonctionne assez rapidement. Nous pourrions aussi utiliser, après approche du taux cherché, la méthode de Newton-Raphson (utilisation de la tangente pour approcher plus vite la valeur recherchée) pour accélérer les choses, mais ce n'est pas très utile dans ce cas précis; vous verrez que la méthode est bien assez rapide, et très simple à programmer.

Etape 1 : Calcul du taux de rentabilité

1) Placer i (taux à calculer) entre 0 et 1. Retenir cet intervalle;

2) calculer l'équilibre obtenu par ce taux avec la formule vue plus haut;

3) si le résultat est supérieur à zéro, enlever la moitié supérieure de l'intervalle, sinon enlever la moitié inférieure (c'est cela, la dichotomie);

4) replacer `i` au milieu du nouvel intervalle;

5) tant que l'équilibre obtenu n'est pas suffisamment proche de zéro, recommencer l'opération à partir du calcul en (2).

6) fin du travail.

Pour cette étape, nous devrons utiliser quelques variables : deux pour retenir l'intervalle, une pour retenir la valeur de i, une autre pour recueillir le résultat du calcul.

Nous n'avons pas encore terminé, loin de là. En effet, le calcul de l'équilibre mérite lui aussi une petite étude en profondeur.

Etape 2 : Calcul de l'équilibre avec un taux i

1) Résultat=F0 pour partir du flux initial F0.

2) Pour `j=1` jusqu'à 12 :

3) calculer `Fj / (1+i)^j`

4) ajouter ce nombre au résultat

5) Fin du travail.

Enfin, le langage C dispose, en standard, d'une fonction permettant de calculer les puissances car c'est un langage civilisé. Mais pour la beauté de l'exercice, nous n'allons pas l'uiliser. Nous allons donc devoir implémenter cette fonction qui apparaît en ligne (3) de l'étape ci-dessus.

Pour cela il existe essentiellement deux méthodes, dont l'une ne fonctionne qu'avec des exposants entiers,et l'autre avec des exposants quelconques.

Etape 3 : Calcul du nombre x à la puissance entière p (p>=2)

Méthode 1 : multiplications successives

1) Résultat=x

2) Pour exposant=2 jusqu'à p

3) Multiplier Resultat par x

4) Fin du travail.

Etape 3 : Calcul du nombre x à la puissance p,

Méthode 2 : Logarithmes

1) Resultat= exponentielle de (p * Log(x))

2) Fin du travail.

Notez que la méthode 1 est très rapide pour les exposants petits car la multiplication et l'addition sont beaucoup plus performantes que le calcul des logarithmes et exponentielles. Par contre, si l'exposant vaut par exemple 200, la seconde méthode sera sans aucun doute plus avantageuse car elle ne procède qu'à un calcul de logarithme, une multiplication, et une exponentielle, au lieu de 200 multiplications.

A ce stade, nous avons terminé l'analyse, car tout est programmable en C. A moins que vous ne souhaitiez reprogrammer les fonctions logarithme et exponentielle, bien sûr ! Mais nous n'irons pas jusque là...

Résumé de l'analyse

Nous avons donc l'analyse suivante :

PROGRAMME PRINCIPAL :
- Stocker par un moyen quelconque les différents flux,F0 jusqu'à F12.
- Calculer le taux de rentabilité interne inconnu, `i`.

Pour cela :
- Placer `i` (taux à calculer) entre 0 et 1. Retenir cet intervalle.
- calculer l'équilibre obtenu par ce taux avec la formule vue plus haut.

Pour cela :
- `Resultat=F0` pour partir du flux initial F0.
- Pour `j=1` jusqu'à 12 :
- calculer `Fj / (1+i)^j`

CALCUL DE LA PUISSANCE :
- `Résultat=x`
- Pour `exposant=2` jusqu'à `p`
- Multiplier `resultat` par `x`
- Fin du travail.

AUTRE CALCUL POSSIBLE A LA PLACE :
- `Résultat=exp(y*ln(x))`
- Fin du travail

- ajouter ce nombre au `resultat`
- Fin du travail.
- si le résultat est supérieur à zéro, enlever la moitié supérieure de l'intervalle, sinon enlever la moitié inférieure.
- replacer `i` au milieu du nouvel intervalle.
- tant que l'équilibre obtenu n'est pas suffisamment proche de zéro, recommencer l'opération à partir du calcul en (2).
- Fin du travail.
- Procurer le résultat
- Fin du travail et du programme !

Vous pouvez remarquer qu'en très peu de temps, nous avons obtenu une analyse assez poussée du problème, et nous pouvons maintenant le programmer.

Pour cela, il nous reste encore à définir quelques petites choses.

STRUCTURE DU PROGRAMME, VARIABLES ET REMARQUES

Pour simplifier, nous allons tout d'abord décider que chaque étape détaillée dans notre analyse sera contenue dans une fonction indépendante. C'est d'ailleurs ainsi que nous avons progressé : après tout, le calcul de la puissance n'a rien à voir avec le reste du programme, et l'algorithme de la dichotomie n'est pas spécifiquement lié au calcul de l'équilibre. Par conséquent, autant tout séparer.

D'autre part, nous utiliserons des variables locales chaque fois que cela sera possible. Souvenez-vous : une variable locale (par exemple locale à une fonction, ou à bloc exécutoire suivant une instruction for), n'est pas accessible en dehors de son domaine; on ne risque ainsi aucun conflit avec l'extérieur.

Nous allons toutefois nous autoriser une variable globale : le tableau contenant les flux. En effet, ce tableau concerne l'ensemble du programme (sauf la fonction puissance, qui sera placée "au dessus" pour ne pas pouvoir l'utiliser) et nous n'allons donc pas nous amuser à la fournir à chacune des fonctions. Ce tableau se nommera FLUX, ou plus exactement `flux` (en minuscules, comme pour toutes les variables, n'oublions pas les conventions du C)

Rappelons que l'emploi des variables globales est franchement déconseillé, sauf si ces variables jouent le rôle de constantes : dans ce cas, en effet, il est pratique de les déclarer en globales pour en faciliter l'accès et éviter de passer en paramètres des constantes à chaque fonction. C'est le cas ici : le tableau des flux, une fois rempli (c'est-à-dire avant tout calcul), n'est plus du tout modifié dans le reste du programme.

Par esprit de simplicité, nous initialiserons le tableau des flux avec des valeurs d'exemple, afin d'éviter une saisie par l'utilisateur. Nous pourrons toujours ajouter plus tard une fonction pour remplir au clavier ce tableau.

Enfin, nous n'avons pas encore traité des problèmes de précision. Il faut bien arrêter la dichotomie à un moment ou à un autre. Pour cela nous allons utiliser deux variables :

1) La première contiendra la précision souhaitée. La fin du calcul sera détectée lorsque la valeur absolue du calcul de l'équilibre sera inférieure ou égale à cette variable.

2) La seconde sera un compteur de boucles, qui sera incrémenté à chaque découpage de l'intervalle de recherche. Ce compteur sera également surveillé, afin de stopper la recherche, si une valeur maximale (choisie par le

programmeur) est atteinte : dans ce cas, on suppose en effet que le programme (ou le compilateur, ou la machine...) ne peut pas trouver le taux avec la précision demandée, sans doute trop fine par rapport à la précision maximale utilisable pour les nombres. Il vaut mieux alors stopper la dichotomie, sans quoi le programme tournerait probablement à l'infini.

On appelle ce type de compteur un "disjoncteur de sécurité". Notez qu'il n'intervient en aucun cas dans l'algorithme ou la résolution du problème : il est uniquement là "pour le cas où", afin d'éviter une boucle infinie due au manque de précision du matériel informatique utilisé pour les calculs. Notez aussi qu'il ne sera utile que si la précision maximale est effectivement trop faible par rapport à celle demandée, mais que dans tous les cas, quelle que soit la machine, quel que soit le compilateur, quelle que soit la précision maximale disponible, il fournira toujours le résultat le plus fin possible. Notez enfin qu'il n'interrompra pas le calcul si la précision recherchée est atteinte avant le nombre maximal de boucles .

Enfin, nous allons nous offrir le luxe d'inclure au programme, grace à une fonction spécifique à Turbo C, un petit chronomètre pour calculer le temps pris par les calculs, et nous exécuterons le calcul pour trois précisions différentes, afin d'observer l'effet de ce paramètre sur le nombre de boucles effectuées.

LA PROGRAMMATION

Maintenant que nous avons tous les éléments, il nous faut penser à programmer. Nous allons ici le faire dans l'ordre chronologique, c'est-à-dire que nous allons examiner les blocs et fonctions au fur et à mesure de leur apparition dans le programme.

Si vous avez bien suivi notre propos, vous savez par quelle fonction nous allons commencer.

Il s'agit de la fonction puissance, bravo ! En effet, c'est la fonction la plus basse dans notre analyse. Voici donc la partie du programme qui lui correspond :

```
double puissance(double x,int p)
{
  int        i;
  double     r = x;
  for (i = 2; i <= p; i++)
  r *= x;     return(r);
}
```

Ceci n'attire aucun commentaire exagérément long. Notez l'utilisation de l'opérateur `*=` dans l'instruction de la boucle `for` : la ligne équivaut à l'instruction d'affectation `r = r * x`, rappelons-le.

Notez également que nous évitons la multiplication par 1 en initialisant `r` (qui reçoit le résultat) avec `x` lors de sa déclaration et en commençant la boucle à l'indice 2.

Ensuite, nous devons penser à créer le tableau des flux. En effet, les fonctions suivantes du programme y feront référence. Nous plaçons ici un exemple :

```
double   flux[13]={ 1019131.0, /* Janvier flux initial */
        0.0,    /* rien en Fevrier     */
        0.0,    /* rien en Mars        */
        0.0,    /* rien en Avril       */
        0.0,    /* rien en Mai         */
        -200000.0,    /* Retrait Juin        */
        0.0,    /* rien en Juillet     */
        0.0,    /* rien en Aout        */
        0.0,    /* rien en Septembre */
        -20000.0,    /* Retrait Octobre     */
        -230000.0,   /* Retrait Novembre    */
        0.0,    /* rien en Decembre   */
        -662139.0};  /* Somme finale (retranchée) */
```

Dans cet exemple, il n'existe aucun dépôt, si l'on excepte celui de Janvier qui constitue en fait le fonds de départ, encore appelé "flux initial". Notez enfin que la dernière case contient un flux imaginaire, qui est le complément des douze autres afin que la somme totale donne 0. N'oublions pas en effet que nous cherchons le taux procurant l'équilibre. Si nous désirions connaitre le taux de rendement pour qu'il reste par exemple 20000 francs sur le compte, il suffirait de remplacer cette valeur par -642139.0.

Ensuite, nous devons programmer la fonction qui calcule justement l'équilibre obtenu avec un taux donné (rappelez-vous, la belle formule !).

Pour cela, nous allons utiliser l'instruction `for` dans sa toute-puissance (et encore, ce n'est qu'un aperçu), puisqu'elle va contenir pratiquement la totalité de la fonction.

Voici la fonction de calcul correspondant :

```
double calcul(double interet)
{
  double   resultat=flux[0];    /* Pour recevoir provisoirement*/
                                  /*le résultat                    */
  int      i;                   /* compteur des mois */

  for (i=1; i<=12 ;
        resultat += flux[i]/puissance(1.0 + interet,i),i++ );
  return(resultat);
 }
```

Dans la boucle for, notez le `i++` qui apparaît en second après la modification de `resultat`. Nous aurions pu placer ce `i++` au sein du calcul lui-même, par exemple en faisant :

```
resultat += flux [i]/puissance(1.0 + interet,i++)
```

Malheureusement, `i` apparaît deux fois dans le calcul, et nous ignorons lequel des deux sera utilisé en premier. Selon le compilateur, en effet, l'interprétation des calculs diffère. Autant séparer la post-incrémentation. Ainsi sreons-nous certains du bon fonctionnement de l'instruction.

Nous aurions également pu écrire ceci :

```
for (i=1 ; i<=12 ; i++)
resultat += flux[i]/puissance(1.0 + interet,i);
```

C'est exactement équivalent. Mais en faisant l'essai, on s'aperçoit que cette méthode demande une quinzaine d'octets en plus, et sur de longs calculs (avec beaucoup d'indices), elle est légèrement plus lente. Les mystères de la compilation...

La fonction suivante va concerner la dichotomie. Nous allons l'appeler "trouve_taux", puisque c'est exactement ce qu'elle fait.

Sa composition est toujours très simple : elle se borne essentiellement à appeler "calcul" et à surveiller la précision et le disjoncteur de sécurité.

```
double trouve_taux(double precision,int sec_intr)
{
  int   compteur=1;     /* compteur initial de l'interrupteur */
  double  i=0.5;        /*  taux de départ */
  double  i_min=0.0;     /* début intervalle de recherche */
  double  i_max=1.0;     /* fin intervalle de recherche */
  double  resultat;     /* reçoit le résultat du calcul de l'équilibre
* /

  d o
  {
    resultat = calcul(i);  /* calcule l'équilibre pour i */
    if (resultat<0)        /* equilibre négatif : augmenter i */
    {
      resultat = -resultat;/* inverser le signe pour test précision */
      i_min=i;              /* suprimer moitié inférieure */
      i=(i+i_max)/2;        /* nouvelle approximation de i */
    }
    else                     /* equilibre positif : diminuer i */
    {
      i_max=i;              /* suprimer moitié supérieure */
      i=(i_min+i)/2;        /* nouvelle approximation de i */
    }
    compteur++;        /* mise à jour compteur de l'interrupteur */
  }
  while( (resultat>precision) && (compteur<sec_intr) );
  printf("%1.16f précision, %d boucles\n",resultat,compteur);
  return(i);
}
```

Maintenant, nous devons appeler cette fonction. Pour cela nous pouvons bien sûr l'appeler directement à partir de `main()`, mais nous allons encore écrire une fonction intermédiaire, qui se chargera de chronométrer la durée de calcul. Cela rendra le programme principal encore plus clair. Toutefois, pour utiliser les fonctions chronomètres aimablement fournies par Borland, il faut d'une part inclure le fichier standard DOS.H, d'autre part connaître les fonctions en questions.

```
void au_boulot(double prec)
{
  int     maxloop=1000;    /* sécurité après 1000 boucles */
  struct time chrono;      /* le chronomètre, merci Borland ! */
  double   taux;           /* récupère le taux calculé */
  int     ecart;           /* pour les chronomètres */

  gettime(&chrono);
  ecart = 60 * chrono.ti_min + chrono.ti_sec;
  taux = trouve_taux(prec,maxloop);
  gettime(&chrono);
  ecart -= 60 * chrono.ti_min + chrono.ti_sec;
  printf("Resultat %1.16f trouvé en %d seconde(s).\n\n",
         taux,-ecart);
}
```

Notez que cette fonction est du type `void`, c'est à dire qu'elle ne renvoie aucun résultat. Ce type appartient au standard ANSI tout récent. Il ne modifie pas l'appel de la fonction, ni grand chose à son fonctionnement, mais a l'avantage d'une part d'éviter la réservation de mémoire pour le retour du résultat lorsque la fonction est appelée, et d'autre part de signaler une éventuelle erreur lors de la compilation si la fonction est utilisée quelque part dans une affectation.

Enfin, il nous reste à écrire le corps principal, la fonction `main()`. Et cette fonction, vous pouvez le constater, est réellement simple :

```
main()
{
  puts("TURBO-C 1.0, Calcul Taux Rentabilité Interne\n");
  au_boulot(1.0e-5);
  au_boulot(1.0e-8);
  au_boulot(1.0e-10);
}
```

Voilà qui clôt la programmation.

On ne peut pas vraiment dire que ce soit compliqué : regardez bien, et vous constaterez que les fonctions sont toutes composées de moins de trente lignes (et encore, avec un grand nombre de vides). Et chacune est indépendante des autres, sauf dans l'ordre défini par notre analyse.

En résumé, nous avons le schéma suivant :

MAIN dépend de AU_BOULOT.

AU_BOULOT dépend de TROUVE_TAUX

TROUVE_TAUX dépend de CALCUL

CALCUL dépend de PUISSANCE

PUISSANCE dépend de ... Turbo C uniquement !

Donc, en programmant les fonctions en sens inverse, nous sommes certains de limiter les bugs au strict minimum. On pourrait encore compacter le tout dans une grande fonction `main()`, car chaque fonction n'est utilisée qu'une seule fois.

CONCLUSION

Comme nous venons de le dire, chaque fonction de ce programme exemple n'est appelée qu'une seule fois par une autre. On peut donc, à la limite, se demander quelle est l'utilité de disloquer ainsi le programme. Somme toute, cela complique la structure du programme. En réalité, vous l'avez sans doute compris, nous voulions mettre en évidence un point important de la programmation moderne.

Jadis, lorsqu'ils travaillaient avec peu de mémoire et des langages assez fatigants à taper car utilisés sur des machines non-interactives, les informaticiens ont inventé le Fortran et ses subroutines, au fonctionnement équivalent aux fonctions du C. Le but d'une subroutine était double :

- Elle évitait de taper deux fois le même bout de programme.

- Elle diminuait la taille du programme, donc son temps de compilation, et sa durée de mise au point.

De nos jours, de tels arguments sont dépassés. En effet, les éditeurs sont plein-écran, faciles à utiliser; ils disposent de copie de blocs de texte, et la taille mémoire n'est plus un problème. La question est alors la suivante : pourquoi utiliser des subroutines puisqu'elles ne se justifient plus ?

Elles se justifient toujours, justement, mais par des arguments beaucoup plus actuels. En effet, il est évident que la durée de mise au point d'un programme est proportionnelle au fouillis qui y règne. Plus un bloc destiné à une tâche particulière comporte de lignes, plus sa mise au point est difficile, puisqu'il comporte de nombreuses instructions dépendantes les unes des autres.

Faire des fonctions, même pour ne les utiliser qu'une seule fois, nous venons de le constater, permet :

- De limiter la complexité de chaque fonction à une tâche bien particulière et facile à programmer, donc à mettre au point.

- De faciliter la récupération de ces fonctions pour d'autres programmes puisqu'elles doivent être indépendantes et former chacune un tout.

- De faciliter la lecture des listings, puisque chacune d'entre elles, dans l'idéal, n'occupe qu'une ou deux pages écran et, si possible, une seule feuille d'imprimante.

De tout cela, il résulte :

- Accélération de la vitesse de développement des programmes.

- Facilité de mise au point et donc de maintenance ou d'évolution.

- Repos du programmeur qui n'a plus à se concentrer sur un ensemble de 200 lignes de programme où se situe une seule erreur : Ici, il n'y a que 20 ou 30 lignes, 50 au maximum.

- Aisance de l'analyse : reportez-vous un peu plus haut pour le constater, l'analyse d'un problème découle d'une démarche naturelle et claire, parfois même intuitive.

La conclusion de tout cela, c'est que plus vos programmes seront propres et structurés, plus vous irez loin.

Malgré une opinion hélas! répandue, la programmation structurée n'est pas destinée à enfermer l'informaticien dans un carcan de règles, mais bel et bien à lui permettre d'utiliser de façon plus efficace ses connaissances et son expérience, et donc à étendre ses horizons. Elle donne paradoxalement un rôle beaucoup plus important à son intuition et à son style, puisque les règles qui la définissent sont simples et facilement assimilées.

Toutefois, il faut convenir que la programmation structurée, qui découle pourtant d'une démarche naturelle, nécessite un certain temps d'adaptation. Mais l'investissement de départ (c'est à dire.. du temps et de l'opiniatreté) est très viteet largement rentabilisé.

Voici pour terminer le listing complet de notre programme :

```
/*************************************************/
/*   TAUX.C                                      */
/*   PROGRAMME CALCULANT LE TAUX DE RENTABILITE  */
/*   INTERNE                                     */
/*************************************************/

/*************************************************/
/* Prototypes                                     * /
/*************************************************/

double puissance(double x,int n);
double calcul(double interet);
double trouve_taux(double precision,int sec_intr);
void au_boulot(double prec);

/*************************************************/
/* pour inclure les fonctions chronomètre TURBO C  */
/*************************************************/

#include <dos.h>

/*************************************************/
/* fonction puissance indépendante               */
/*************************************************/

double puissance(double x,int n)
{
  int   i ;
  double  p  =  x;

  for (i = 2; i <= n; i++)
    p  *=  x;
  return(p);
}

/*************************************************/
/* déclaration/initialisation du tableau des flux  */
/*************************************************/

double  flux[13]={ 1019131.0,     /* Janvier flux initial */
              0.0,                /* rien en Fevrier    * /
              0.0,                /* rien en Mars     * /
              0.0,                /* rien en Avril     * /
              0.0,                /* rien en Mai       * /
         -200000.0,               /* Retrait Juin      * /
              0.0,                /* rien en Juillet    * /
              0.0,                /* rien en Aout      * /
              0.0,                /* rien en Septembre    * /
```

```
          -20000.0,          /* Retrait Octobre    * /
         -230000.0,          /* Retrait Novembre   * /
              0.0,           /* rien en Decembre   * /
         -662139.0};         /* Somme finale retranchée */

/*************************************************/
/* Fonction de calcul de l'équilibre             */
/*************************************************/

double calcul(double interet)
{
 double  resultat=flux[0];  /* Pour recevoir provisoirement*/
                            /*le résultat */
 int   i;                   /* compteur des mois */
 for (i=1; i<=12 ; i++)
  resultat += flux[i]/puissance(1.0 + interet,i);
 return(resultat);
}

/*************************************************/
/* Fonction effectuant la dichotomie             * /
/* Controle la précision obtenue et l'interrupteur */
/* de sécurité                                     */
/*************************************************/

double trouve_taux(double precision,int sec_intr)
{
 int   compteur=1;     /* compteur initial de l'interrupteur */
 double  i=0.5;        /*  taux de départ */
 double  i_min=0.0;    /* début intervalle de recherche */
 double  i_max=1.0;    /* fin intervalle de recherche */
 double  resultat;     /* reçoit le résultat du calcul de
                          /*l'équilibre */

 d o
 {
  resultat = calcul(i);  /* calcule l'équilibre pour i */
  if (resultat<0)        /* equilibre négatif : augmenter i */
  {
   resultat = -resultat;/* inverser le signe pour test précision */
   i_min=i;              /* suprimer moitié inférieure */
   i=(i+i_max)/2;        /* nouvelle approximation de i */
  }
  else                   /* equilibre positif : diminuer i */
  {
   i_max=i;              /* suprimer moitié supérieure */
   i=(i_min+i)/2;        /* nouvelle approximation de i */
  }
```

```
    compteur++;              /* mise à jour compteur de l'interrupteur */
  }
  while( (resultat>precision) && (compteur<sec_intr) );
  printf("%1.16f précision, %d boucles\n",resultat,compteur);
  return(i);
}

/*****************************************************/
/* fonction effectuant un calcul chronométré         */
/*****************************************************/

void au_boulot(double prec)
{
  int     maxloop=1000;     /* sécurité après 1000 boucles */
  struct time chrono;       /* le chronomètre, merci Borland ! */
  double   taux;             /* récupère le taux calculé */
  int     ecart;             /* pour les chronomètres */

  gettime(&chrono);
  ecart = 60 * chrono.ti_min + chrono.ti_sec;
  taux = trouve_taux(prec,maxloop);
  gettime(&chrono);
  ecart -= 60 * chrono.ti_min + chrono.ti_sec;
  printf("Resultat %1.16f trouvé en %d seconde(s).\n\n", taux,-ecart);
}

/*****************************************************/
/* Le programme principal lui même (minuscule)       */
/*****************************************************/

main()
{
  puts("TURBO-C 1.0, Calcul Taux Rentabilité Interne\n");
  au_boulot(1.0e-5);
  au_boulot(1.0e-8);
  au_boulot(1.0e-10);
}
```

ANNEXE B

LES OPERATEURS

Niveau	Forme	Signification	Associativité
15	()	expression,	-->
15	[]	indexation	-->
15	->	membre de structure	-->
14	!	non logique	<--
14	~	"non" binaire	<--
14	++	incrémentation	<--
14	--	décrémentation	<--
14	-	inversion de signe	<--
14	(type)	casting	<--
14	*ptr	élément pointé	<--
14	&var	pointeur	<--
14	sizeof	taille	<--
13	*	multiplication	-->
13	/	division	-->
13	%	modulo	-->
12	+	addition	-->
12	-	soustraction	-->
11	<<	décalage à gauche	-->
11	>>	décalage à droite	-->
10	<	Comparaison inf	-->
10	>	Comparaison sup	-->
10	<=	Comparaison inf-egal	-->
10	>=	Comparaison sup-egal	-->
9	==	Egalité	-->
9	!=	Différence	-->
8	&	ET binaire	-->
7	^	XOR binaire	-->
6	|	OU binaire	-->
5	&&	ET logique	-->
4	||	OU logique	-->
3	?:	Exp. conditionelle	-->
2	=	Affectation	<--
2	+=	Affectation	<--
2	-=	Affectation	<--
2	*=	Affectation	<--
2	/=	Affectation	<--
2	%=	Affectation	<--
2	|=	Affectation	<--
2	^=	Affectation	<--
2	&=	Affectation	<--
2	>>=	Affectation	<--
2	<<=	Affectation	<--
1	,	Séparateur	-->

ANNEXE C

TABLE ASCII

DEC	HEX	CAR
32	20	
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L

DEC	HEX	CAR
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y

DEC	HEX	CAR
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	^
128	80	Ç
129	81	ü
130	82	é
131	83	â
132	84	ä
133	85	à
134	86	å
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	ì
142	8E	Ä
143	8F	Å
144	90	É
145	91	æ
146	92	Æ
147	93	ô
148	94	ö
149	95	ò
150	96	û
151	97	ù
152	98	ÿ
153	99	Ö
154	9A	Ü
155	9B	ç
156	9C	£
157	9D	¥
158	9E	₧
159	9F	ƒ
160	A0	á
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	Ñ
166	A6	ª

DEC	HEX	CAR
167	A7	º
168	A8	¿
169	A9	⌐
170	AA	¬
171	AB	½
172	AC	¼
173	AD	¡
174	AE	«
175	AF	»
176	B0	▒
177	B1	▓
178	B2	│
179	B3	┤
180	B4	╡
181	B5	╢
182	B6	╖
183	B7	╕
184	B8	╣
185	B9	║
186	BA	╗
187	BB	╝
188	BC	╜
189	BD	╛
190	BE	┐
191	BF	└
192	C0	┴
193	C1	┬
194	C2	├
195	C3	─
196	C4	┼
197	C5	╞
198	C6	╟
199	C7	╚
200	C8	╔
201	C9	╩
202	CA	╦
203	CB	╠
204	CC	═
205	CD	╬
206	CE	╧
207	CF	╨
208	D0	╤
209	D1	╥
210	D2	╙
211	D3	╘

DEC	HEX	CAR
212	D4	╒
213	D5	╓
214	D6	╫
215	D7	╪
216	D8	┘
217	D9	┌
218	DA	▒
219	DB	▓
220	DC	▒
221	DD	▓
222	DE	▒
223	DF	▓
224	E0	α
225	E1	ß
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	µ
231	E7	τ
232	E8	Φ
233	E9	Θ
234	EA	Ω
235	EB	δ
236	EC	∞
237	ED	φ
238	EE	ε
239	EF	∩
240	F0	≡
241	F1	±
242	F2	≥
243	F3	≤
244	F4	⌠
245	F5	⌡
246	F6	÷
247	F7	≈
248	F8	°
249	F9	∙
250	FA	·
251	FB	√
252	FC	ⁿ
253	FD	²
254	FE	■
255	FF	

ANNEXE D

BIBLIOGRAPHIE

Les ouvrages recommandés ci-dessous sont classés par niveau de difficulté de lecture (ou niveau de connaissance et de pratique requis) croissante.

OUVRAGES SUR LE LANGAGE C:

Turbo C pas à pas

J.M. Gaudin

Editions PSI

Un trés bon petit livre pour les tout-débuts en Turbo C. Style très vivant, agréable à lire, sans prétention, un ouvrage à conseiller à tout ceux qui ne connaissent absolument rien au C ni au Pascal. Sa lecture nécessite bien entendu un approfondissement par d'autres ouvrages, car il ne va pas très loin, mais il est parfait pour se mettre en appétit.

Standards, style et exercices en C

Michel de Champlain

Editions Dunod

Un excellent manuel pratique à consulter et qui décrit tout ce qui concerne la mise en oeuvre du C. Très propre, et agréable à lire. A considérer comme un manuel à garder à portée de main car son organisation et son format sont vraiment très pratiques.

Le langage C

B.W. Kernighan et D.M. Ritchie

Editions Masson

La "bible", par les auteurs du langage eux-mêmes. Un peu obsolete par rapport aux possibilités de Turbo C, mais garanti 100% compatible avec tous les compilateurs, donc de toute façon indispensable pour tous les problèmes de portabilité. Relativement peu lisible pour un débutant (style discutable) mais sans problème dès le premier pas franchis.

Variations en C

Steve Schustack

Editions Cedic-Nathan/Microsoft Press

Un beau livre rempli de programmes et de bonnes méthodes, agréable à lire, très penché sur la mise en œuvre du C sur des problèmes concrets, très utile pour le débutant et intéressant pour le connaisseur. Bon niveau, progressif et suffisamment poussé.

OUVRAGES SUR L'ASSEMBLEUR ET MS-DOS :

L'assembleur du 8088 et de l'IBM PC (et compatibles)

David C. Willen et Jeffrey I. Krantz

Editions Mémoire vive (diffusé par PSI)

Un livre procédant d'une manière originale pour découvrir l'assembleur et le système. Facile à lire, progressif, mais vraiment très original par rapport aux autres démarches, peut être un peu déconcertant. En tout cas très illustré, et sujet bien traité.

Assembleur 8088 et BIOS IBM-PC

Bruno Le Maire et Gilles Mauffrey

Editions EdiTests/PSI

Un livre pour commencer l'assembleur, très progressif et traitant les principaux problèmes. Beaucoup d'exemples, dont une partie utilisable sans le Macro-Assembleur de Microsoft (avec DEBUG livré dans MS-DOS). Un bon livre sur le sujet.

MS-DOS avancé

Ray Duncan

Microsoft Press (pour l'instant en Anglais)

Excellent livre sur la mise en œuvre de MS-DOS dans les règles de l'art. L'architecture du DOS et l'utilisation en respectant les règles de portabilité et d'évolutivité est traitée sous toutes les coutures, de même que la mise en

œuvre de toutes les ressources de l'IBM PC: écran, clavier, mémoire, disques, ports... Un ouvrage indispensable. Notons également la présence de plusieurs exemples en C et en assembleur (deux versions pour comprendre mieux les différences de travail).

MANUELS DE REFERENCES TECHNIQUES SUR LE SYSTEME IBM PC/AT:

Au coeur de l'IBM PC, logique et fonctionnement interne

J.B. Thiele

Editions Editests

Nous conseillons particulièrement cet ouvrage à ceux qui n'ont aucune idée de la manière dont fonctionne leur ordinateur, et qui veulent se pencher sur ce fonctionnement interne pour pratiquer l'assembleur ou le C au niveau système. L'ouvrage est une bonne présentation de la machine, et permet d'avoir assez rapidement une idée précise sur le rôle du BIOS, des contrôleurs, du DOS, etc... Libre au lecteur ensuite d'abandonner ou de se procurer un manuel plus facile à consulter. Mais cet ouvrage remplit parfaitement sa fonction d'apprentissage.

Clefs pour PC et compatibles/ Clefs pour PC-AT et compatibles

Daniel Martin et François Piette

Editions PSI

Un bon manuel résumant l'ensemble des caractéristiques des PC sous un format pratique et une organisation structurée. Liste des fonctions DOS, BASIC, accès aux différents circuits, fonctions du BIOS, etc... A garder non loin du clavier ! On regrette les pages inutiles sur le Basic, qui auraient pu être consacrées à une illustration des fonctions du MS-DOS ou du BIOS : les lecteurs de ce style d'ouvrages ne programment plus forcément en Basic interprété ! Mais c'est un très bon manuel. A recommander à ceux qui savent ce qu'ils cherchent.

8086/8088, Programmation en assembleur

François Retaureau

Editions Sybex

Un des nombreux ouvrages "bible" sur le processeur 8088/8086. De tous ceux-ci, il nous a semblé le mieux fourni et le mieux organisé. Mais c'est sans doute subjectif. Il pousse l'étude du sujet jusqu'à l'utilisation des macros sous

MASM, des fonctions DOS et BIOS, etc. Un ouvrage très fourni, qu'on ne regrette pas si l'on désire se pencher sur l'assembleur de façon assez poussée. Peu destiné aux débutants.

ALGORITHMIQUE ET PROGRAMMATION STRUCTUREE:

Introduction à la programmation (2 tomes)

J. Biondi et G. Clavel

Editions Masson

L'inévitable, l'incontournable ouvrage des étudiants en Informatique, une excellente présentation des techniques algorithmiques de base. Ensuite, au lecteur de jouer !

ANNEXE E

INDEX

THEMES :

FONCTIONS DE LA LIBRAIRIE

Votre avis nous intéresse

Pour nous permettre de faire de meilleurs livres, adressez-nous vos critiques sur le présent ouvrage.

— ***Titre de votre livre :*** .....

— ***Ce livre vous donne-t-il toute satisfaction ?***

— ***Y a-t-il un aspect du problème que vous auriez aimé voir abordé ?***

Si vous souhaitez des éclaircissements techniques, écrivez-nous, ou laissez un message à notre service Minitel (3615 OI PSI), nous ne manquerons pas de vous répondre directement.

Avez-vous déjà acquis des livres P.S.I. ?

lesquels ? .....

.....

qu'en pensez-vous ?

.....

Nom ..... Prénom ..... Age .....

Adresse .....

Profession .....

Centre d'intérêt .....

Je désire recevoir un catalogue gratuit et être informé régulièrement des nouveaux livres publiés.

☐ oui ☐ non

Veuillez renvoyer cette page, dûment remplie, aux

Editions PSI
5, place du Colonel Fabien
75491 PARIS CEDEX 10

Achevé d'imprimer
sur les presses de l'imprimerie IBP
à Rungis (Val-de-Marne 94) (1) 46.86.73.54
Dépôt légal - Novembre 1987

N° d'impression: 4952
N° d'édition: 86595-467-1
N° d'ISBN: 2-86595-467-6

TURBO-C

P.S.I.

LANGAGES

LE LIVRE DE TURBO-C SUR PC ET COMPATIBLES

Turbo-C est en train de s'affirmer comme l'un des compilateurs C les plus vendus du marché grâce à sa rapidité de compilation, sa simplicité d'utilisation et son excellent rapport qualité/prix.

Le livre de Turbo-C sur PC et compatibles vous apprend à maîtriser les différents éléments indispensables à la programmation en Turbo-C. Destiné plus spécialement au programmeur découvrant ce langage, il vous mènera progressivement à un haut niveau de connaissance.

Après la présentation générale de la philosophie de Turbo-C et la description de son installation sur PC et AT et de son environnement de programmation, vous ferez vos premiers pas en découvrant successivement les différents types de variables, les tableaux, les instructions de base, etc. Une deuxième étape vous familiarisera avec des notions plus évoluées : structuration, modularité, visibilité des variables, les pièges, les pointeurs et les fichiers.

PRIX : 145 FF

ISBN : 2-86595-467-6

9 782865 954674

ÉDITIONS P.S.I.
DIFFUSÉ PAR P.C.V. DIFFUSION
BP 86 - 77402 LAGNY-S/MARNE CEDEX – FRANCE